High-Throughput Three-Party DPFs with Applications to ORAM and Digital Currencies

Guy Zyskind
guyz@mit.edu
MIT Media Lab, Fhenix
Cambridge, MA, USA

Avishay Yanai
avishay@sodalabs.xyz
Soda Labs
Tel Aviv, Israel

Alex "Sandy" Pentland
pentland@mit.edu
MIT Media Lab
Cambridge, MA, USA

ABSTRACT

Distributed point functions (DPF) are increasingly becoming a foundational tool with applications for application-specific and general secure computation. While two-party DPF constructions are readily available for those applications with satisfiable performance, the three-party ones are left behind in both security and efficiency. In this paper we close this gap and propose the first three-party DPF construction that matches the state-of-the-art two-party DPF on all metrics. Namely, it is secure against a malicious adversary corrupting both the dealer and one out of the three evaluators, its function's shares are of the same size and evaluation takes the same time as in the best two-party DPF. Compared to the state-of-the-art three-party DPF, our construction enjoys $40 - 120\times$ smaller function's share size and shorter evaluation time, for function domains of $2^{16} - 2^{40}$, respectively.

Apart from DPFs as a stand-alone tool, our construction finds immediate applications to private information retrieval (PIR), writing (PIW) and oblivious RAM (ORAM). To further showcase its applicability, we design and implement an *ORAM with access policy*, an extension to ORAMs where a policy is being checked before accessing the underlying database. The policy we plug-in is the one suitable for *account-based* digital currencies, and in particular to central bank digital currencies (CBDCs). Our protocol offers the first design and implementation of a large scale privacy-preserving account-based digital currency. While previous works supported anonymity sets of 64-256 clients and less than 10 transactions per second (tps), our protocol supports anonymity sets in the millions, performing $\{500, 200, 58\}$ tps for anonymity sets of $\{2^{16}, 2^{18}, 2^{20}\}$, respectively.

Toward that application, we introduce a new primitive called *updatable DPF*, which enables a direct computation of a dot product between a DPF and a vector; we believe that updatable DPF and the new dot-product protocol will find interest in other applications.

1 INTRODUCTION

A *point function* $f_{\alpha,\beta}$ is a function that evaluates to zero everywhere, except for one point, α, at which it evaluates to a (possibly) non-zero value β. A *distributed point function* (DPF), introduced by [6, 28], is a cryptographic technique to share a point function to multiple receivers, such that each receiver can locally obtain a share of the evaluation of the function at the point of interest. In the two receiver setting, a DPF scheme allows one to generate two additive shares of $f_{\alpha,\beta}$, called f_0 and f_1, such that for every x in the domain it holds that $f_0(x) + f_1(x) = f(x)$. The performance of a DPF scheme is measured by the size of the shares (also known as 'keys') f_i given to the receivers and the incurred computational complexity to evaluate $f(x)$ for some x in the domain. In the two-party case (i.e., when the point function is shared toward two receivers) the shares size as well as the time to evaluate are logarithmic in the domain size.

The DPF primitive is increasingly recognized as a fundamental tool in various applications, including private information retrieval (PIR), anonymous communication, privacy-preserving machine learning (PPML) and even as a building block for general secure computation for RAM programs [4, 17, 18, 31, 39, 42, 46, 50, 54]. Having said that, the DPF constructions today are practical in the setting of two receivers, therefore, most of the applications mentioned above were demonstrated efficient when employing a two-party construction of a DPF; leaving the challenge of extending them to a larger number of parties unlocked.

A natural and important milestone is to efficiently extend DPFs to three receivers with honest majority. While several works have addressed this setting, none of them reached performance that matches that of the two receivers.[1] In this work we present the first DPF construction with three-receivers that accomplishes that goal, and as shortly explained, *even surpasses* the performance of the two receivers setting.

Database operations via DPFs. Most of the applications that use DPFs as building blocks focus on either reading from or writing to a database. Specifically, consider the 'replicated database' private information retrieval (PIR) setting with a (public) database D. In that setting, a client may privately read the α-th database entry by sharing a point function $f_{\alpha,1}$, and sending the shares f_0 and f_1 to the PIR servers. Then, the i-th server computes $d_i = \sum_x f_i(x) \cdot D[x]$ and hands d_i back to the client. Finally, the client can reconstruct the result by computing $d = d_0 + d_1$. Note that evaluating $f(x)$ over the entire domain $|D|$ creates a shared one-hot-vector, so d correctly encodes $D[\alpha]$. Obviously, this procedure does not work when the database itself is secret shared.

On the other hand, when the database is secret shared among the two servers, i.e., there are two databases D_0 and D_1 s.t. $D[x] = D_0[x] + D_1[1]$, a DPF can be used in order to privately *update* an entry in that database (also known as private information writing, or PIW). Specifically, a client sends the shares f_0 and f_1 for some point function $f_{\alpha,\beta}$, and then the i-th server adds $f_i(x)$ to the value at $D[x]$, for all x in the domain. This way, the client can privately add β to the α-th entry of the shared database.

Supporting both private information retrieval and writing operations, however, is more challenging, as privately reading an entry requires multiplication between the shared entries of the one-hot vector to those of the database, which is achieved via communication in the information theoretic setting, or via expensive public-key

[1]The works in [48, 56] achieve a three-party protocol with a matching performance, however, we still consider it in the two receivers domain as one of the parties plays as a non-colluding server.

primitives; both approaches limit the size of the databases that a system may support. For example, the FLORAM system by Doerner and Shelat [23] requires $O(\sqrt{|D|})$ communication per private operation (read or write).

In contrast, given a DPF construction for three receivers, where sharing $f_{\alpha,\beta}$ results in f_1, f_2, f_3 s.t. $f_1(x)$, $f_2(x)$ and $f_3(x)$ form a 2-out-of-3 sharing of $f(x)$, we can efficiently achieve both private retrieval and update operations. This is due to the fact that the shares now have redundancy and have one degree of multiplicative homomorphism. The only work that proposed such a construction is [10], where the resulting shares form a replicated sharing. However, their construction achieves sub-optimal performance, both concretely and asymptotically. That is, their shares size as well as the computational complexity per evaluation is $O(\log^2 |D|)$. Furthermore, it does not protect against a malicious adversary. In this work, we propose the first three party DPF construction that matches (and even concretely improves) the two party one; additionally, it offers security against a malicious adversary. On its own, our three party DPF lends itself as the main tool for our efficient three-server ORAM, which has state-of-the-art performance compared to previous DPF-based ORAM constructions. Also, to the best of our knowledge, ours is the only maliciously secure construction.

Revisiting private and account-based digital currencies. The UTXO (Unspent Transaction Output) model (as used by Bitcoin) tracks individual unspent coins from previous transactions. In contrast, the account model (as used by Ethereum) resembles traditional banking, with each user having an account and a clear balance. Transactions adjust these balances directly, simplifying operations for complex programs like smart contracts.

Existing works on account-based privacy-preserving digital currencies have typically been constrained by a limited anonymity set size [12, 14, 15, 22, 26, 41]. Consequently, the focus has predominantly been on the UTXO model, as evidenced by the plethora of works ([3, 5, 38, 47, 52, 53] to name a few). However, the UTXO model has several significant shortcomings. For example, it limits programmability and auditability [34, 35], which are crucial for building financial applications and fraud-prevention. Additionally, in the privacy-preserving setting, the UTXO set infinitely grows, which poses scalability issues.

In light of this, using our DPF-based ORAM construction, we aim to rekindle interest in account-based privacy-preserving digital currencies, addressing its inherent limitations and proposing a viable alternative to private UTXO-based systems. We place a particular emphasis on the application of our techniques to Central Bank Digital Currencies (CBDCs). Given their rising prominence [9, 35, 44, 53, 59] and the ongoing debate around their privacy [2, 3, 52, 58], our goal is to demonstrate the potential of our methods in developing a privacy-centric CBDC. Such a system would not only safeguard individual privacy but also remain conducive to necessary auditability.

1.1 Our Contributions

We summarize our main contributions below.

- We present a novel three-party verifiable DPF (VDPF) construction in the honest majority setting that is secure against a malicious server who may collude with the client. Our construction maintains the same asymptotic and concrete overheads (share size and evaluation time) as the state-of-the-art two-party DPF of Boyle et al. [7]. Compared to the state-of-the-art three-party DPF, our construction improves the DPF's shares size and evaluation by a factor of $O(\log n)$ asymptotically, translating to $48-120\times$ improvement for domains between $2^{16}-2^{40}$.
- As the lion share of the overhead in a three-server ORAM is incurred by the DPF, we get the most efficient DPF-based three-server ORAM construction, in both communication and computation. We analytically compare ourselves against state of the art three-party protocols using DPF and ORAM [10, 11, 20, 48, 55]. We analyze all with respect to a semi-honest ORAM application and observe that we have better client-server communication, overall computation and storage costs, and our model does not have the limitations of [48, 56] (server-aided), and [20] (writes are not supported; only appends). Our comparison results are summarized in Table 1 [2][3].
- Expanding on our main construction, we introduce a three-party VDPF with semi-honest servers security, that uses a sublinear amount of PRF calls in full-domain evaluation. This construction is $\sim$ 2.2× faster than the leading two-party DPF during full-domain evaluation, with potential for further improvement using GPU acceleration due to CPU-optimized AES instructions. However, similar to two-party DPFs, this construction is confined to separate read or update operations and is not suitable as a complete ORAM construction.
- Motivated by the need for privacy-preserving account-based digital currencies [2, 3, 52, 58], we apply our DPF-based ORAM to a three-party private CBDC protocol. Our implementation demonstrates substantial improvements in throughput over prior works in the account model. Specifically, our implementation supports 500, 200 and 58 (resp. 825, 300 and 105) transactions per second (tps) for anonymity sets of 2^{16}, 2^{18}, and 2^{20} accounts, with protection against a malicious adversary (resp. a semi-honest adversary). We compare to the previous state-of-the-art, Solidus [14], and find that for these settings our protocol's throughput is $11-141\times$ higher.
- One building block in our maliciously secure DPF construction is a newly introduced primitive we call *updatable VDPF* (or UVDPF). An efficient UVDPF construction is used in order to directly compute a dot-product between (the compressed) full-domain evaluation of a point function and a vector (or of two point functions). Our dot-product protocol is constant-round and incurs communication overhead that is sub-linear in the functions' domain (compared to linear in a naive implementation). We believe that the UVDPF primitive and the DPFs dot-product protocol will find interest in other applications as well.

1.2 Related Work

DPF Constructions and three-party ORAM. As observed by previous works [10, 11, 20], $(2,3)$-DPFs are better suited for applications that combine PIR with PIW (ORAM), as they offer one degree of

[2] For readability, we ignore constants related to the security parameter or the output group size.
[3] We estimate our results generalize to DORAM as well, but we leave exploring that to future work.

Protocol	Key Size	Read	Update	DB Copies	Malicious	Model
[10]	$3\sqrt{n}$	$4n$	$4n$	2	No	3-party
[11]	$12\log^2 n$	$O(n\log n)$	$O(n\log n)$	2	No	3-party
[20]	$3\log n$	$2n$	Appends only	2	Yes	3-party
[48, 56]	$3\log n$	$2n$ + Interaction	$3n$	3	No	Server-aided
Ours	$2\log n$	$2n$	$2n$	1	Yes	3-party

Best	Neutral	Poor	Worst

Table 1: Comparison of three-party ORAM protocols using DPFs. Key Size captures client-server communication, Reads and Updates are non-interactive (except for [48, 56]) and focus on overall computation costs. Results are for semi-honest, but we remark that we and [20] provide malicious implementations.

multiplicative homomorphism. Also, all $(3,3)$-DPFs (withstanding two corruptions) are significantly less efficient, as they require $O(\sqrt{n})$-sized keys, and either have much higher constants or require public-key (or lattice-based) operations [1, 7, 10, 18, 42].

The state-of-the-art for $(2,3)$-DPF [11, 20, 48, 56] combines replicated secret sharing (RPSS) with $(2,2)$-DPFs, and then uses these to build different ORAM-related applications. Bunn et al., [11] proposes a $(2,3)$-DPF similar to us, but their construction has asymptotically larger keys and evaluation time, which makes it $48-120\times$ less efficient, in both computation and communication, for domains between $2^{16}-2^{40}$ (and this gap widens further for larger domains). Alternatively, Waldo, Duoram and PRAC [20, 48, 56] take a different approach and use (2,2)-DPFs to construct their ORAM application (a private time-series database and DORAM respectively), but their solutions are more limited as they do not develop new dedicated three-party DPFs. Waldo cannot do writes or updates (only appends), while Duoram and PRAC operate in the server-aided model. Only Waldo offers malicious security as we do, but they rely on public access control, rather than the privacy preserving authentication that our protocol offers.

A different line of work focused on *verifiable DPFs*, essentially achieving security for DPFs against potentially malicious clients (dealers). [4, 7] suggested to use sketching techniques, and more recently [21] presented a lightweight solution based on hashing. Similarly, [42, 49] show how to do private access-control for DPFs. We similarly build verifiability and access-control capabilities into our (2,3)-DPF construct. Our construction protects against both malicious clients and servers, whereas prior work only protected against semi-honest servers.

Privacy-preserving digital currency. A large body of work has been dedicated to ensuring transaction-privacy in blockchains. The majority of works [5, 13, 27, 38, 40, 45, 47, 53] focused on the UTXO-model, which offers limited programmability and auditability [34], add end-user and developer complexity [35, 47], and cause an infinite growth of the permanent database (the nullifiers). This led to the transition into the account-based model (e.g., Ethereum), which we follow in this paper. Solutions in this model mostly build upon general MPC techniques that enable the addition of additional layers on top of it (like smart contracts, auditability, etc.) [24, 33, 61, 62].

The challenge for account-based ledgers with private transactions. Solving privacy for account-based cryptocurrencies remains a challenge, and prior work has significant limitations compared to UTXO-based solutions. QuisQuis [26] and Zether [12, 22] operate in the account-model but are limited to extremely low k-anonymity sets (64-256) [4]. Very recently, [37] presented a theoretical solution using Fully Homomorphic Encryption (FHE), but they neither provide an implementation or an evaluation. The heavyweight cryptography used indicates it might not be concretely efficient.

An increasing body of research is focused on constructing CBDCs [3, 9, 35, 44, 52, 53, 59]. Yet, to the best of our knowledge, we are the first to provide a private account-based solution. Most similar to our research, several works have proposed building private bank-to-bank cryptocurrencies (e.g., [14, 15, 41]). In these works, banks transact with each other on behalf of users, but the number of banks in all of these models is very small (10-100). While Solidus is more scalable, zkLedger and MiniLedger are geared more towards auditability. We compare against Solidus which is the state-of-the-art and show that our work has $11-141\times$ higher throughput up to an anonymity set of 2^{20}. Additionally, while our protocol provides full anonymity, the rest of these works only provides bank-level anonymity, which is orders of magnitude smaller (equal to the total number of banks). Conversely, compared to these other systems, our system is not publicly verifiable and relies on MPC assumptions.

2 PRELIMINARIES

Basic notation. We use κ as a computational security parameter. For $x, y \in \{0,1\}^*$ the expression $x||y$ is the concatenation of x and y. Uniformly sampling a random value x from a set X is denoted by $x \overset{\$}{\leftarrow} X$. The result of a probabilistic algorithm A on inputs $x_1, x_2, \ldots$ is written by $x \leftarrow A(x_1, x_2, \ldots)$; in addition, when we want to explicitly mention the randomness used in the algorithm we write $x = A(x_1, x_2, \ldots; r)$. Vectors are formatted in bold, e.g., $\boldsymbol{x}$, and the i-th index is denoted by $x^{(i)}$. The identity vector $\boldsymbol{e}_i$ has zero in all coordinates except in its i-th coordinate, which equals one. Also, for better readability, when describing our CBDC construction, we denote vectors in uppercase, e.g., X instead.

2.1 Distributed Point Functions

We start off by presenting a definition for DPFs.

[4] Estimates from [37]

Definition 2.1. *Let $\alpha \in \{1, \ldots, n\}$ and $\beta \in \mathbb{F}$, a $(2,2)$-Distributed Point Function (DPF), denoted $F^{(2,2)}_{\alpha,\beta}$ is defined by algorithms:*

- $(f_0, f_1) \leftarrow \mathsf{DPF.Gen}(1^\kappa, \alpha, \beta)$.
- $y_b = \mathsf{DPF.Eval}(b, f_b, x)$, *where* $b \in \{0,1\}$.

Correctness. It must hold that $\mathsf{DPF.Eval}(0, f_0, \alpha) + \mathsf{Eval}(1, f_1, \alpha) = \beta$ *and* $\mathsf{DPF.Eval}(0, f_0, x) + \mathsf{DPF.Eval}(1, f_1, x) = 0$ *for all* $x \neq \alpha$.

Privacy. For every $b \in \{0,1\}$ there exists a simulator $\mathcal{S}$ such that

$$f_b \overset{c}{\equiv} \mathcal{S}(1^\kappa, b, n)$$

where $(f_0, f_1) \leftarrow \mathsf{DPF.Gen}(1^\kappa, \alpha, \beta)$ *and the distribution is over the coin tosses of algorithms* $\mathsf{DPF.Gen}$ *and* $\mathcal{S}$.

Evaluation can also be defined on a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, by:

$$\begin{aligned}\boldsymbol{y}_b &= \mathsf{DPF.Eval}(b, f_b, \boldsymbol{x}) \\ &= (\mathsf{DPF.Eval}(b, f_b, x_1), \ldots, \mathsf{DPF.Eval}(b, f_b, x_n)).\end{aligned}$$

By the correctness of the DPF, it holds that $\boldsymbol{y} = \boldsymbol{y}_0 + \boldsymbol{y}_1 = \boldsymbol{e}_\alpha \cdot \beta$. When $\boldsymbol{x}$ covers the entire domain of α this is called a *full domain evaluation.* The computational complexity of a full domain evaluation is better than an individual evaluation on each $x_i \in \boldsymbol{x}$ in isolation.

The most efficient $(2,2)$-DPF was developed by Boyle et al. [7]. It has keys (f_0 and f_1) of length $O(\log n)$, where n is the size of the domain and a full domain evaluation incurs $O(\kappa \cdot n)$ computation. We note that the scheme in [7] describes the DPF over a general group $\mathcal{G}$ whereas in the constructions presented in this paper we use either the binary field $\mathbb{F}_{2^l}$ where $l = \kappa$ or a prime field $\mathbb{F}_q$ defined with the prime $q \approx 2^\kappa$. For simplicity we denote the prime field by $\mathbb{F}$, leaving the prime parameter implicit. Addition of $x, y \in \mathbb{F}_{2^l}$ is computed by $x \oplus y$ and addition of $x, y \in \mathbb{F}$ is computed by $x + y \mod q$.

We continue with a definition of *verifiability*, which enables the key holders to verify the validity. Formally:

Definition 2.2. *A verifiable DPF (VDPF) is a DPF (as per Definition 2.1) with the following additional procedures:*

- $\pi_b = \mathsf{VDPF.Prove}(b, f_b)$, *where* $b \in \{0,1\}$.
- $\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathsf{VDPF.Verify}(\pi_0, \pi_1)$

Correctness is the same as in Definition 2.1; Privacy is enhanced to include the proof generated by the other key, that is, there exists a simulator $\mathcal{S}$ such that

$$(f_b, \pi_{1-b}) \overset{c}{\equiv} \mathcal{S}(1^\kappa, b, n)$$

where $(f_0, f_1) \leftarrow \mathsf{VDPF.Gen}(1^\kappa, \alpha, \beta)$ *and* $\pi_b = \mathsf{VDPF.Prove}(b, f_b)$, *where $b \in \{0,1\}$ and the distribution is over the coin tosses of algorithms* $\mathsf{VDPF.Gen}$ *and* $\mathcal{S}$.

Verifiability. This ensures that the two keys are well formed*, that is: Let* $\boldsymbol{y}_b = \mathsf{VDPF.Eval}(b, f_b, \boldsymbol{x})$, *then* $\mathtt{accept} = \mathsf{VDPF.Verify}(\pi_0, \pi_1)$ *if and only if* $\boldsymbol{y}_0 + \boldsymbol{y}_1 = \boldsymbol{e}_{\alpha'} \cdot \beta'$ *for some* α', β'.

2.2 Shamir Secret Sharing

Secret sharing enables a dealer to split a secret x into n *shares*, such that only a sufficiently large subset of shares can be used to recover the secret. Due to space considerations we only focus on the notation here.

- $[x] = \mathsf{SS.Share}(x; r)$. Given a secret $x \in \mathbb{F}$ and a random tape r, pick $a_1, \ldots, a_t \in \mathbb{F}$ and output $[x] = \{[x]_1, \ldots, [x]_n\}$, where $[x]_i = P(i)$ for a degree-t polynomial $P(\mathsf{x}) = x + a_1\mathsf{x} + a_2\mathsf{x}^2 + \ldots + a_t\mathsf{x}^t$ (note the difference between x, the secret, and x, the polynomial variable).
- $x = \mathsf{SS.Reconstruct}([x]_{i_1}, \ldots, [x]_{i_{t+1}})$. Given $t+1$ shares $[x]_{i_1}, \ldots, [x]_{i_{t+1}}$, where $1 \leq i_1 < i_2 < \ldots < i_{t+1} \leq n$, interpolate a polynomial P such that $P(i_j) = [x]_{i_j}$ for all $j \in [1, t+1]$ and output $x = P(0)$ if interpolation succeeds, otherwise output $\perp$.

2.3 ORAM with Policy Verification

ORAM, introduced by Goldreich and Ostrovsky [29] and followed by many works ([36, 51, 57, 60] to name a few), is a cryptographic protocol between a client and a server. Such a protocol enables the client to upload its database to a server and access specific records in that database while being protected from the server. Besides the usual confidentiality guarantees (which are typically enabled by encryption schemes) an ORAM guarantees the client that nothing about the *access pattern* is being leaked to the server; that is, the server learns nothing about the entries being accessed or the actions being applied to those entries (either read or update). As shown time and again (e.g., [30]), such access pattern leakage may be devastating for a system that strives to preserve the client's privacy. Over the years, there were protocols that proposed to distribute the server's role into multiple (distrustful) entities in order to improve efficiency and to support more than one client (where sections of the database may be accessed by many clients). Such solutions are known as *multi-client multi-server ORAM*. In this work we propose a multi-client three-servers ORAM with an extended functionality we call *policy verification*. Specifically, the ORAM is parameterized with a policy function that is triggered whenever an access is requested by a client, and returns 'accept' or 'reject'. The access request is executed only if the policy returns 'accept'.

In Figure 1 we present our extended functionality for an ORAM with access-policy, called $\mathcal{F}_{\mathsf{AP-ORAM}}$. Notice that in many applications (including the one in this work) it is required to hide the content of a request, but not its type, and so the $\mathcal{F}_{\mathsf{AP-ORAM}}$ functionality exposes different commands to these two types: read and update. Typically, when the operation type itself is to be hidden, protocols apply a 'read-then-update' mechanism, where both commands are invoked one after the other (with a 'dummy' read or a 'dummy' update argument), which preserves obliviousness for that too. This is not required for the application presented in this paper and so it remains out of scope.

The functionality exposes a 'register' command, in which the client is assigned with an index of a record that it can access. In the realization of that functionality, after a client has registered it may access the functionality without being identified; namely, whether a client's request is approved for execution or not depends only on the content of the request, which may include some hidden authentication information. Further notice that the functionality reflects the fact that when considering a malicious adversary it grants the adversary with the ability to abort the execution of requests. The parts related to a malicious adversary are colored in gray. The read and update commands exposed by the functionality support a read

Setting: The functionality interacts with servers S_1, S_2, S_3, clients $c_1, \ldots, c_n$, and an adversary $\mathcal{S}$.

Parameters: The functionality is initialized with a zero-initialized array $D = (D_1, \ldots, D_N) \in \mathbb{F}^N$, and is parameterized with a policy verification function: PVerify that is given the request's arguments and returns `accept` or `reject`. Parameter m refers to the number of entries read/update commands support.

- On input $(c_i, \text{register})$ from client c_i, mark c_i as 'registered'.
- On input $(c_i, \text{read}, (\ell_1 \ldots, \ell_m))$ from a registered client c_i:
  - Send (read, m) to $\mathcal{S}$ and wait to its response; if $\mathcal{S}$ returns abort then send $\perp$ to $\mathcal{S}$ and all servers, otherwise (if $\mathcal{S}$ returns continue) continue.
  - If accept = PVerify$(c_i, \text{read}, (\ell_1 \ldots, \ell_m))$ then output $(D_{\ell_1}, \ldots, D_{\ell_m})$ to c_i.
- On input $(c_i, \text{update}, (\ell_1, v_1), \ldots, (\ell_m, v_m))$ from a registered c_i:
  - Send (update, m) to $\mathcal{S}$ and wait to its response; if $\mathcal{S}$ returns abort then send $\perp$ to $\mathcal{S}$ and all servers, otherwise (if $\mathcal{S}$ returns continue) continue.
  - If accept = PVerify$(c_i, \text{update}, ((\ell_1, v_1), \ldots, (\ell_m, v_m)))$ then, update $D_{\ell_j} = D_{\ell_j} + v_j$ for every $j \in \{1, \ldots, m\}$.

Figure 1: Functionality $\mathcal{F}_{\text{AP-ORAM}}$

and update of m entries in bulk, where m is a parameter. For the update command we assume that the locations $\ell_1, \ldots, \ell_m$ to update are distinct.

2.4 Basic Functionalities

We use of the following ideal functionalities:

- $\mathcal{F}$.Rand(), $\mathcal{F}$.DRand(), and $\mathcal{F}$.Zero(), which return a shared random value, shared random with degree t and $2t$, and a sharing of zero, respectively.
- $\mathcal{F}$.A2B($[x]$). Converts an arithmetic shamir sharing to a binary Shamir sharing.
- $\mathcal{F}$.Mult($[x], [y]$). Returns a (t, n)-Shamir sharing of the product $x \cdot y$.
- $\mathcal{F}$.SoP($[\boldsymbol{x}], [\boldsymbol{y}]$). Returns a (t, n)-Shamir sharing of the dot-product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$. Note that this functionality is semi-honest, and in the malicious case, the adversary can add an additive error to the output.
- $\mathcal{F}$.CheckZero($[x]$). Returns 1 if $[x] = 0$, or 0 otherwise.
- $\mathcal{F}$.LTE($[x], [y]$). Returns 1 if $[x] \leq [y]$ and 0 otherwise.

These functionalities are well established in the literature and so we omit further implementation details. These can be found in the full version of this paper.

In addition, we define a procedure, *Open*($[x]$) as a one-round (assuming broadcast) protocol where all parties send their shares to each other and reconstruct a secret.

3 (2, 3)-VERIFIABLE DPF

In this section, we formally define the notion of a $(2, 3)$-VDPF, and provide our construction as well as a security proof. To this end, we move away from additive shares and require that the individual evaluations by keys f_1, f_2, f_3 on input α form a valid Shamir sharing of the target value β. This is in contrast to other works [11, 20] where those values form a replicated secret sharing. We start by defining a (2,3)-VDPF:

Definition 3.1. *A $(2, 3)$-VDPF, denoted $F^{(2,3)}_{\alpha,\beta}$, is defined by algorithms:*

- $(f_1, f_2, f_3) \leftarrow \text{VDPF.Gen}(1^\kappa, \alpha, \beta)$ *and*
- $y_b \leftarrow \text{VDPF.Eval}(b, f_b, x)$ *$(b \in \{1, 2, 3\})$,*
- $\pi_b = \text{VDPF.Prove}(b, f_b, r)$, *$(b \in \{1, 2, 3\})$,*
- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \text{VDPF.Verify}(\pi_1, \pi_2, \pi_3)$

such that:

Correctness. Similar to the $(2, 2)$ case, except that we use Shamir's reconstruction rather than mere group addition. Let $y_b(x) = \text{Eval}(b, f_b, x)$, then:

- $\text{SS.Reconstruct}(y_1(x), y_2(x), y_3(x)) = \beta$ *for $x = \alpha$, and*
- $\text{SS.Reconstruct}(y_1(x), y_2(x), y_3(x)) = 0$ *for all $x \neq \alpha$.*

Privacy. For every $b \in \{1, 2, 3\}$ there exists a simulator $\mathcal{S}$ such that

$$(f_b, \pi_{b'}, \pi_{b''}) \overset{c}{\equiv} \mathcal{S}(1^\kappa, b, n)$$

where $\{b, b', b''\} = \{1, 2, 3\}$, $(f_1, f_2, f_3) \leftarrow \text{VDPF.Gen}(1^\kappa, \alpha, \beta)$ and the distribution is over the coin tosses of algorithms VDPF.Gen *and $\mathcal{S}$.*

Verifiability. Let $\boldsymbol{y_x} = \text{SS.Reconstruct}(y_1(x), y_2(x), y_3(x))$, then $\texttt{accept} = \text{VDPF.Verify}(\pi_1, \pi_2, \pi_3)$ *iff $\boldsymbol{y} = \boldsymbol{e}_{\alpha'} \cdot \beta'$ for some α', β'.*

We note that in the above definition β as well as the outputs of algorithm Eval are drawn from a prime field $\mathbb{F}$.

In Section 3.2 we present our construction for $(2, 3)$-VDPF, which uses $(2, 2)$-VDPF$^+$ from Section 3.1 as a building block. In Section 3.3 we provide a security proof for our VDPF construction.

3.1 Building Block: $(2, 2)$-VDPF$^+$

One tool we use in order to construct our $(2, 3)$-VDPF is a $(2, 2)$-VDPF$^+$ (or an *enhanced* VDPF); its non-verifiable version was introduced in [11] and our addition of verifiability is straightforward, assuming we use (2,2)-VDPFs of [21] internally.

Definition 3.2. *A $(2, 2)$-VDPF$^+$, denoted $F^{(2,2)}_{\alpha,\beta_0,\beta_1}$, is a $(2, 2)$-VDPF, as defined in Section 2.1, with the following additional constraint: It must hold that* $\text{VDPF.Eval}(b, f_b, \alpha) = \beta_b$ *for $b \in \{0, 1\}$.*

The additional constraint ensures that at the special point α, party $b = 0$ receives β_0 and party $b = 1$ receives β_1. In Figure 2 we provide the construction for $F^{(2,2)}_{\alpha,\beta_0,\beta_1}$ from [11], given a 'normal' DPF construction. Note that the resulting VDPF is a normal VDPF for parameters α, β, where all individual evaluations are shifted by z. Then, obviously since both parties XOR their evaluation with z, this does not change the combined evaluation on any point (as it simply adds $z \oplus z = 0^l$; while on point $x = \alpha$, it causes f_0's (resp. f_1's) evaluation be β_0 (resp. β_1), which is easy to verify.

3.2 Our $(2, 3)$-VDPF Construction

Our $(2, 3)$-VDPF construction is formally given in Figure 3.

We use two independent instantiations of a $(2, 2)$-VDPF$^+$ (Section 3.1) and output three keys f_1, f_2, f_3 where f_i is a pair of $(2, 2)$-VDPF$^+$ keys, one from each instance. Specifically, denote the first and second $(2, 2)$-VDPF$^+$ keys by (g_0, g_1) and (k_0, k_1), respectively, then our three $(2, 3)$-VDPF keys are $f_1 = (g_0, k_0)$, $f_2 = (g_1, k_0)$ and $f_3 = (g_1, k_1)$. Evaluating f_i on input x is done by first interpreting it as the keys g and k of a $(2, 2)$-VDPF$^+$; then, evaluating g and k independently on x and adding the results (over $\mathbb{F}_{2^l}$).

The $(2, 2)$-VDPF$^+$ instances above are adjusted so that evaluating g_0 and g_1 on α results with v_0 and v_1, respectively; similarly, evaluating k_0 and k_1 on α results with v_2 and v_3, respectively. It

```
VDPF⁺.Gen($1^\kappa, \alpha, \beta_0, \beta_1$)
    (1) $\beta \leftarrow \beta_0 \oplus \beta_1$
    (2) $(f_0, f_1) \leftarrow$ VDPF.Gen($1^\kappa, \alpha, \beta$)
    (3) $z \leftarrow \beta_0 \oplus$ VDPF.Eval($0, f_0, \alpha$)
    (4) $f_b \leftarrow (z, f_b)$ for $b \in \{0, 1\}$
    (5) Output: $(f_0, f_1)$.

VDPF⁺.Eval($b, f_b, x$)
    (1) $(z, f'_b) \leftarrow f_b$.
    (2) $y \leftarrow$ VDPF.Eval($b, f'_b, x$) $\oplus z$.
    (3) Output: $y$

VDPF⁺.Prove($b, f_b$)
    (1) $(z, f'_b) \leftarrow f_b$.
    (2) $\pi'_b \leftarrow$ VDPF.Prove($b, f'_b$).
    (3) Output $\pi_b = (\pi'_b, z)$

VDPF⁺.Verify($\pi_0, \pi_1$)
    (1) Output accept iff $\pi_0 = \pi_1$.
```

Figure 2: Protocol for $(2, 2)$-VDPF$^+$

is required that the three values (1) $\beta_1 = v_0 \oplus v_2$, (2) $\beta_2 = v_1 \oplus v_2$ and (3) $\beta_3 = v_1 \oplus v_3$ be valid Shamir sharing of the target value $\beta_0 = \beta$; namely, that there is a degree-1 polynomial P s.t. $P(i) = \beta_i$ for $i \in \{0, 1, 2, 3\}$. Fixing a random Shamir sharing $(\beta_1, \beta_2, \beta_3)$ of β, equations (1)-(3) above always have a solution (assignments to v_0, v_1, v_2, v_3); moreover, since the sharing is random, by drawing a random v_0 we get that the two values obtained by evaluating f_i (i.e., $(v0, v_2)$, (v_1, v_2) and (v_1, v_3)) are distributed uniformly in $\mathbb{F}_{2^l} \times \mathbb{F}_{2^l}$.

A subtle issue in the construction is that the shares $\beta_1, \beta_2, \beta_3$ are from a prime field $\mathbb{F}$ whereas v_0, v_1, v_2, v_3 are from $\mathbb{F}_{2^l}$, thus, equations (1)-(3) above do not 'compile'. To reconcile that, we define the operation $\boxplus$ that takes an element x from a prime field $\mathbb{F}$ and an element y from a binary field $\mathbb{F}_{2^l}$, 'embeds' x into $\mathbb{F}_{2^l}$ by simply using its binary representation to form x', and outputs $x' \oplus y$. In addition, for x and y as above, we define the operation $\odot$ that embeds y into $\mathbb{F}$ to form y' and outputs $x \cdot y'$ over $\mathbb{F}$.

For these operations to work as expected, and to not raise a security concern, we must have that the binary representation of an element in $\mathbb{F}$ be well defined over $\mathbb{F}_{2^l}$ and that the arithmetic representation of an element in $\mathbb{F}_{2^l}$ be well defined over $\mathbb{F}$. This is not true in general, however, we can pick a prime and binary fields for which the above almost always holds. Concretely, using $l = \kappa$ and $\mathbb{F}$ with a prime very close to 2^κ will achieve the desired result. In this manner, we get $\frac{2^\kappa - |\mathbb{F}|}{2^\kappa} \approx 2^{-\kappa}$ and so values are drawn (either at random or as a result of a computation) from the gap between the fields only with negligible probability.

The above establishes that evaluation of keys f_1, f_2, f_3 on input $x = \alpha$ results with a Shamir sharing of the target value β. For completeness, we now show that evaluation on every other input (i.e., $x \neq \alpha$) results with a Shamir sharing of zero. Since the $(2, 2)$-VDPF$^+$ is defined over a binary field $\mathbb{F}_{2^l}$, evaluation of g_0 and g_1 (respectively of k_0 and k_1) on $x \neq \alpha$ results with the same value (so adding them results with 0^l). Thus, evaluation of $f_i = (g_{i_g}, k_{i_k})$ ($i \in \{1, 2, 3\}, i_g, i_k \in \{0, 1\}$) results with the same value for all

```
Parameters: A prime field $\mathbb{F}$ and a hash function $H$. Inverses are
computed over the prime field $\mathbb{F}$.
VDPF.Gen($1^\kappa, \alpha, \beta$)
    (1) $(\beta_1, \beta_2, \beta_3) \leftarrow \mathsf{SS}_{2,3}$.Share($\beta$)
    (2) $\beta'_i \leftarrow \beta_i \cdot (i)^{-1}$, for $i \in \{1, 2, 3\}$.
    (3) $v_0 \overset{\$}{\leftarrow} \mathbb{F}_{2^l}$
    (4) $v_2 \leftarrow \beta'_1 \boxplus v_0$
    (5) $v_1 \leftarrow \beta'_2 \boxplus v_2$
    (6) $v_3 \leftarrow \beta'_3 \boxplus v_1$
    (7) $(g_0, g_1) \leftarrow$ VDPF⁺.Gen($1^\kappa, \alpha, v_0, v_1$)
    (8) $(k_0, k_1) \leftarrow$ VDPF⁺.Gen($1^\kappa, \alpha, v_2, v_3$)
    (9) Set $f_1 = (g_0, k_0)$, $f_2 = (g_1, k_0)$ and $f_3 = (g_1, k_1)$
   (10) Output: $(f_1, f_2, f_3)$.
VDPF.Eval($b, f_b, x$)
    (1) Parse $(g, k) \leftarrow f_b$.
    (2) Set $(b_g, b_k)$ to $(0, 0)$, $(1, 0)$ or $(1, 1)$ if $b$ equals $1, 2$ or $3$, re-
        spectively.
    (3) Compute $y_g$ = VDPF⁺.Eval($b_g, g, x$)
    (4) Compute $y_k$ = VDPF⁺.Eval($b_k, k, x$)
    (5) Compute $y = (y_g \oplus y_k) \odot b$.
    (6) Output $y$.
VDPF.Prove($b, f_b, r_b$)
    (1) Parse $(g, k) \leftarrow f_b$.
    (2) Set $(b_g, b_k)$ to $(0, 0)$, $(1, 0)$ or $(1, 1)$ if $b$ equals $1, 2$ or $3$, re-
        spectively.
    (3) $\pi_g$ = VDPF⁺.Prove($b_g, g$) and $\pi_k$ = VDPF⁺.Prove($b_k, k$)
    (4) Initialize $\boldsymbol{y_b} = \{\}$, $\boldsymbol{u} = \{\}$.
    (5) For $x_i \in \{x_1, \ldots, x_n\}$:
        (a) Compute $y_g(x_i)$ = VDPF⁺.Eval($b_g, g, x_i$)
        (b) Compute $y_k(x_i)$ = VDPF⁺.Eval($b_k, k, x_i$)
        (c) $y_i \leftarrow (y_g(x_i) \oplus y_k(x_i)) \odot b$
        (d) $\boldsymbol{y_b} \leftarrow \boldsymbol{y_b} \cup \{y_i\}$
        (e) Generate (the same) $u_i \overset{\$}{\leftarrow} \mathbb{F}$ and let $\boldsymbol{u} \leftarrow \boldsymbol{u} \cup \{u_i\}$
    (6) $\beta_b \leftarrow \sum_{i=1}^{n} \boldsymbol{y_b}[i]$
    (7) $t_b \leftarrow (\boldsymbol{y_b} \cdot \boldsymbol{u})^2 - \beta_b(\boldsymbol{y_b} \cdot \boldsymbol{u}^2) - r_b$ (over $\mathbb{F}$)
    (8) Output $\pi_b = (\pi_g, \pi_k, t_b, H(t_b))$
VDPF.Verify($\pi_1, \pi_2, \pi_3$)
    (1) Parse $\pi_b = (\pi_{g,b}, \pi_{k,b}, t_b, h_b)$
    (2) Compute $t$ = SS.Reconstruct($t_1, t_2, t_3$)
    (3) Output accept iff $\pi_{g,2} = \pi_{g,3}$ and $\pi_{k,1} = \pi_{k,2}$
        and accept = VDPF⁺.Verify($\pi_{g,1}, \pi_{g,2}$) and accept =
        VDPF⁺.Verify($\pi_{k,1}, \pi_{k,3}$), and $t = 0$ and $\forall i \in \{0, 1, 2\}$
        $H(t_b) = h_b$.
```

Figure 3: Our $(2, 3)$-VDPF construction

i's; let that value be y. Then, an interpolation using the points $(1, y), (2, y), (3, y)$ results with an horizontal line at height y, which leads to the secret y. To get a secret 0 instead, we multiply the result, y, by the index of the key, which now results with the points $(1, y), (2, 2y), (3, 3y)$, resulting with a line that crosses through $(0, 0)$ and so hides the secret 0.

Finally, that multiplication by i requires fixing the target values we give to the VDPF$^+$ instances: instead of working with the shares $\beta_1, \beta_2, \beta_3$, we work with the values $\beta'_1 = \beta_1 \cdot 1^{-1} = \beta_1$, $\beta'_2 = \beta_2 \cdot 2^{-1}$ and $\beta'_3 = \beta_3 \cdot 3^{-1}$.

3.3 Security analysis

Correctness can be verified from the protocol, in the following we prove verifiability and privacy.

THEOREM 3.3. *The construction in Figure 3 is a VDPF (Def. 3.1), secure against a malicious client and a single malicious server.*

PROOF. *Verifiability.* We show that if the verification function outputs `accept` then the functions f_1, f_2, f_3 are well formed (e.g., they evaluate to a Shamir sharing of zero at all points except of at most one, where they evaluate to a non-zero value). Since the VDPF$^+$ outputs `accept` in both verifications, this means that the functions (g_0, g_1) and (k_0, k_1) are well formed (g_0, g_1, k_0 and k_1 have only one non-zero value).

Denote by $y_{g_b}(x)$ (resp. $y_{k_b}(x)$) the result of evaluation of g_b (resp. k_b) at point x. The above guarantees that y_{g_0} and y_{g_1} differ on at most one point. Let that point be α_g and denote the evaluation by $y_{g_0} = \beta_{g,0}$ and $y_{g_1} = \beta_{g,1}$, respectively. Similarly, let α_k be the point at which y_{k_0} and y_{k_1} differ, and denote the evaluation by $y_{k_0} = \beta_{k,0}$ and $y_{k_1} = \beta_{k,1}$, respectively.

We show that if $\alpha_g \neq \alpha_k$ then our verification rejects. Assume $\alpha_g \neq \alpha_k$, that the parties partially exchanged $H(t_b)$, where H is a collision-resistant hash function, before exchanging the rest of their proof (this serves as a commitment and is needed for malicious servers), and that the verification procedure accepted. This implies that the vector $\boldsymbol{y}$ has two non-zero positions, as it is the result of adding (over $\mathbb{F}_{2^l}$) two vectors with a single non-zero value. Assume these values are β_{α_g} and β_{α_k}. From this, we get that the calculation of t in line 7 of Prove yields: $t = (\beta_{\alpha_g} u_{\alpha_g} + \beta_{\alpha_k} u_{\alpha_k})^2 \quad - (\beta_{\alpha_g} + \beta_{\alpha_k}) \left(\beta_{\alpha_g} u^2_{\alpha_g} + \beta_{\alpha_k} u^2_{\alpha_k}\right) = 0$. This is since we assumed the verification accepts. After re-arranging, this equation is simplified to $(u_{\alpha_g} - u_{\alpha_k})^2 = 0$, which implies $u_{\alpha_g} = u_{\alpha_k}$, in contradiction.

Privacy. Per Definition 3.1, each server's view is: $(f_b, \pi_{b'}, \pi_{b''})$, where f_b consists of two different VDPF+ keys, each of which is further composed of a (2,2) VDPF key and a value z. The underlying (2,2) VDPF keys are all indistinguishable from random by the privacy of those VDPF keys.

It is also easy to see that for each VDPF+ key, z is pseudorandom, given that our selection of $v_0, v_1, v_2, v3$ is pseudorandom. However, recall that our construction evaluates two (2,2)-VDPF+ keys, and then adds (over $\mathbb{F}_{2^l}$) their result together. At every point $x \neq \alpha$, we are adding two pseudorandom shares together, but at the point $x = \alpha$, by construction, we obtain $\beta'_b \in \mathbb{F}$. We therefore need to ensure that β'_b is indistinguishable from a random value in $\mathbb{F}_{2^l}$. Otherwise, an adversary controlling b could run a full domain evaluation of its key, and look for a value that is distinguishable from all others (which are pseudorandom in $\mathbb{F}_{2^l}$). This potentially leaks both β'_b and α, so the simulation would fail.

Thus, to conclude the proof, we focus on showing that β'_b is indistinguishable from a random value in $\mathbb{F}_{2^l}$. W.l.o.g., we will assume we are looking at the view of party $b = 0$, which holds β'_0. We note that the analysis for party $b = 1$ and $b = 2$ is similar. First, observe that β'_0 itself is random in $\mathbb{F}$, from the security of Shamir sharing. Then, by our assumption, the probability that a randomly generated value falls in the gap between the two fields is: $\frac{||\mathbb{F}|-2^\kappa|}{2^\kappa} \approx 2^{-\kappa}$, which implies that β'_0 also appears random in $\mathbb{F}_{2^l}$, since $l = \kappa$ as well.

□

4 ACCOUNT-BASED DIGITAL CURRENCY

As mentioned in the introduction, a $(2, 3)$-VDPF provides an efficient way to construct an ORAM protocol. We will now use the construction developed in the previous section to build a protocol that realizes F_{APORAM}. We will specifically focus on an access policy for a privacy-preserving digital currency, in which each user controls a single row in the database of balances (this is equivalent to the account-model of blockchains). We note again that since our solution distributes trust across three servers, we believe it is especially relevant to CBDCs, which are growing in popularity [9, 35, 44, 53, 59]. However, they currently pose significant privacy challenges [2, 3, 52, 58], which our system can overcome.

We formalize the CBDC application using the AP-ORAM functionality (See Functionality 1), with the following parameters: The number of locations to read is $m_{\text{read}} = 1$ and the number of locations to update is $m_{\text{update}} = 2$; these values replace the single parameter m used in the functionality. In addition, the PVerify parameter function is defined by:

$$\text{PVerify}(c_i, \text{read}, \ell_1) = \begin{cases} \texttt{accept} & \text{if } i = \ell_1 \\ \texttt{reject} & \text{otherwise} \end{cases} \tag{1}$$

$$\text{PVerify}(c_i, \text{update}, \ell_1, v_1, \ell_2, v_2) = \begin{cases} \texttt{accept} & \text{if } i = \ell_1 \text{ and } v_1 = v_2 \text{ and } D_i \geq v_1 \\ \texttt{reject} & \text{otherwise} \end{cases} \tag{2}$$

where D_i is part of the state held by the functionality, see Figure 1. That is, the procedures of interest in a CBDC application are "moving" funds from one account to the other (a *transfer*), or reading a balance. Reading a balance is allowed to the account owner and so the function verifies that the client ID (index) matches the row to be read. Moving funds is translated into two memory updates, the first subtract the value in one entry and the second increase the value in another (possibly same, if one is paying itself) entry. It is required that one can pay only from its own account, and so the same matching verification is done as in the balance check procedure; additionally, it is required that the subtracted value at the payer's account and the added value at the payee account are equal. In the rest of the paper, we denote $\mathcal{F}_{\text{AP-ORAM}}$ with the above parameters by $\mathcal{F}_{\text{CBDC}}$.

4.1 Registration & Access Control for DPFs

Recently, Servan-Schreiber et al., [42, 49] presented a mechanism called *private access control lists* (PACL) to verify that a private access to a database (or a vector) D succeeds only if the client requesting that access is permissioned. The privacy of such access, as in our case, is provided via a DPF which enables hiding the location and the value to be written (in case of an update). One drawback of their scheme is that it requires the servers to perform $O(N)$ ($N = |D|$) public-key operations per check, as each server requires to compute an inner-product in-the-exponent. Although their scheme incurs only $O(1)$ communication, performing so many public-key operations per check (in case N is large) is prohibitive.

Instead, we make the observation that since we are in an honest-majority setting, we can modify their protocol to use secret-shared values (instead of values in the exponent) and still compute inner-products with $O(1)$ complexity, and with cheap information-theoretic operations, using the sum-of-products trick (e.g., [1, 63]). We describe the protocol and notation below. Security follows from the same arguments as in [49] and the security of sum-of-products.

The registration procedure, as described in Figure 4, is very simple: each new client receives a random value, λ, from the servers; this random value is known only to the client (and is secret shared at the servers). This value is used by the client each time it wishes to access the servers. The servers maintain a database, denoted Λ, for those secret shared values $[\lambda_1]$, $[\lambda_2], \ldots$, where λ_i is known to client c_i.

Access control for CBDC. In the CBDC application, the i-th entry of the database D 'belongs' to the i-th client, c_i, which means that c_i is the only one who can read it, and decrease the value stored there (up to zero). Suppose that client c_i wants to read D_i (recall that all entries in D are secret shared by the servers). Client c_i now sends a new secret sharing of λ_i, namely $([\lambda_i]_1, [\lambda_i]_2, [\lambda_i]_3)$, to the servers, as well as a VDPF (f_1, f_2, f_3) that encodes the value 1 at location i. This way, the servers can evaluate the DPF to obtain a new shared database, denoted T, that hides 1 at location i and 0 elsewhere. The servers can compute the dot-product between Λ and T, which results with the secret authentication value stored for the client at location i, call that value $\tilde{\lambda}_i$. If the client indeed has access to location i then it must hold that λ_i that is shared by the client at the time of the protocol is equal to $\tilde{\lambda}_i$ that is already stored at location i in Λ. This is a proof that the request sender knows the required authentication secret for some location i that is 'one-hot' encoded in the database T. Then, computing a dot-product between T and D results with the balance b of that exact location that the client proved it knows the authentication secret for.

Initialization. The servers initialize a zero-shared database $D = (D_1, \ldots, D_N)$. Denote the i-th share of the database by $D^i = (D^i_1, \ldots, D^i_N)$. Similarly, the servers maintain a sharing of $\Lambda = (\Lambda_1, \ldots, \Lambda_N)$, which is used for access control. In addition, the servers maintain a zero-initialized counter ctr, which keeps track on the number of registered clients.

(1) **Register.** On input $(c, \text{register})$ to the servers: if $\text{ctr} = N$ then send `full` to the client and halt, otherwise:
  (a) The servers increment ctr and invoke $[\lambda] \leftarrow \mathcal{F}_{\text{Rand}}$; then, each server S_j stores $\Lambda^j_{\text{ctr}} \leftarrow [\lambda]_j$ and sends ctr and $[\lambda]_j$ to the client c. From this point and on, c is indexed c_{ctr}.
  (b) Client computes $\lambda = \text{SS.Reconstruct}([\lambda]_1, [\lambda]_2, [\lambda]_3)$ and stores (ctr, λ).

Figure 4: Protocol Π_{CBDC}.register.

4.2 The CBDC Protocol

The protocol is described in Figures 4-6. If the underlying MPC protocols realizing the well-known functionalities we use (e.g., $\mathcal{F}$.Mult, $\mathcal{F}$.CheckZero, $\mathcal{F}$.Product) are maliciously secure (resp. semi-honest), then our entire protocol protects against malicious servers (resp. semi-honest). Except for the protocol $\mathcal{F}$.Product, we can use semi-honest protocols and turn them maliciously-secure by running them over authenticated inputs (i.e., using MACs) [16, 19], which is exactly what we do in practice. However, doing so for $\mathcal{F}$.Product would require $O(N)$ communication per requrest, which is prohibitively expensive. We solve that problem via a novel maliciously-secure protocol for a dot-product between a DPF and a vector (see Section 5).

The protocol is described in the ORAM language, with Read and Update requests. In the CBDC context, the Read request is actually only used by a client c_i to get its balance. Since the client does not reveal which database entry it reads, yet it proves (using the authentication secret described above) that it is allowed to read that entry, this action is anonymous. This is important, as it reduces the attack surface of network correlation attacks that may exist if requesting the balance would not be anonymous. Then, the Update request is actually only used by a client in order to pay another client. For a client c_i to pay amount v to another client c_j, the client runs the ORAM protocol with inputs (ℓ_1, v_1) and (ℓ_2, v_2) with $\ell_1 = i, \ell_2 = j$ and $v_1 = v_2 = v$. Security against a corrupted server follows from the fact that the underlying building blocks and functionality are secure; a formal simulation based proof is given below. Security against a corrupted client follows the authentication technique described above, which we briefly expound on now. The client sends two VDPFs, each is verifiable on its own and therefore it is guaranteed that each represents a vector with at most one entry that is non-zero. To verify that the two non-zero values are the same, the hidden values are 'extracted' (these are values a and b) and their difference is later handed to $\mathcal{F}$.CheckZero. Note that in contrast to the Read operation, where the client provided a sharing of its authentication secret λ, in Update the client provides a sharing of $\lambda \cdot v_1$ (i.e., the authentication secret times the amount to transfer). Then, the protocol obtains another instance of that value by computing a dot-product between Λ and f (the vector shared by (f_1, f_2, f_3)), and again, the difference between these results is checked using $\mathcal{F}$.CheckZero. Finally, to check that the amount to transfer is no greater than the balance of the user, the protocol computes v' – the user's balance times the amount to transfer, and a_2 the amount to transfer squared. Obviously, it must hold that the latter is no greater than the former, which is checked via a call to $\mathcal{F}_{\text{LTE}}$. We note that because of that check, it is must be ensured that the balances in the system do not exceed the square root of the underlying field size (i.e. $\sqrt{|\mathbb{F}|}$).

Security. An intuition to the security of the protocol was given above; in this section we give a formal simulation-based proof of security. We prove the following:

Theorem 4.1. *Given a verifiable enhanced distributed point function scheme* VDPF$^+$*, protocol* $\Pi = (\Pi_{\text{CBDC}}.\text{register}, \Pi_{\text{CBDC}}.\text{read}, \Pi_{\text{CBDC}}.\text{update})$ *(from Figures 4-6) securely computes* $\mathcal{F}$.CBDC *(Figure 1 with* PVerify *from Equations 1-2) in the* $\mathcal{F}_{\mathsf{X}}$*-hybrid model, for all* $\mathsf{X} \in \{\text{Rand, Zero, Mult, Product, CheckZero, LTE}\}$.

Proof. We show a proof against a malicious adversary. Let $\mathcal{A}$ be an adversary who corrupts server S_c $(c \in \{1, 2, 3\})$ and any subset of the clients. We present a simulator $\mathcal{S}$ that runs $\mathcal{A}$ internally and produce's a simulated adversarial view and output set associated with

Read. On input (read, ℓ) to client c_i:

(1) *Client:* The client c_i sends $(f_i, [\lambda]_i)$ to server S_i, where:
  (a) $(f_1, f_2, f_3) \leftarrow \text{VDPF}^+.\text{Gen}(1^\kappa, \ell, 1)$.
  (b) $([\lambda]_1, [\lambda]_2, [\lambda]_3) \leftarrow \text{SS.Share}_{2,3}(\lambda_i)$.

(2) *Servers:* Given $(f_i, [\lambda]_i)$, the servers invoke $[z] \leftarrow \mathcal{F}.\text{Zero}$; and then, server S_j computes:
  (a) $T^j \leftarrow \text{VDPF}^+.\text{Eval}(j, f_j)$ (this is a full-domain evaluation).
  (b) $[t]_j = \sum_{k=1}^{N} T_k^j$.
  (c) $\pi_j \leftarrow \text{VDPF}^+.\text{Prove}(j, f_j, [z]_j)$.
  (d) $[\tilde{\lambda}] \leftarrow \mathcal{F}.\text{Product}([\Lambda], f)$.

(3) Servers verification:
  (a) Server S_j exchanges h_j from π_j, then exchanges the rest of the proof.
  (b) The servers halt if:
    (i) The DPF is not valid, i.e., reject $=$ $\text{VDPF}^+.\text{Verify}(\pi_1, \pi_2, \pi_3)$,
    (ii) The DPF hides a value different than 1, i.e., $\mathcal{F}.\text{CheckZero}([t] - 1)$ returns false.
    (iii) Client authentication fails, i.e., $\mathcal{F}.\text{CheckZero}([\lambda] - [\tilde{\lambda}])$ returns false.

(4) Balance retrieval:
  (a) The servers compute $[b] \leftarrow \mathcal{F}.\text{Product}([D], f)$ and then server S_j sends $[b]_j$ to the client.

(5) *Client:* The client outputs $b = \text{SS.Reconstruct}([b]_1, [b]_2, [b]_3)$.

Figure 5: Protocol Π_{CBDC}.read.

Update. On input $(\text{update}, (\ell_1, v_1), (\ell_2, v_2))$ to client c_i:

(1) *Client:* The client c_i sends $(f_i, g_i, [\lambda]_i)$ to server S_i, where:
  (a) $(f_1, f_2, f_3) \leftarrow \text{VDPF}^+.\text{Gen}(1^\kappa, \ell_1, v_1)$.
  (b) $(g_1, g_2, g_3) \leftarrow \text{VDPF}^+.\text{Gen}(1^\kappa, \ell_2, v_2)$.
  (c) $([\lambda]_1, [\lambda]_2, [\lambda]_3) \leftarrow \text{SS.Share}_{2,3}(\lambda_i \cdot v_1)$.

(2) *Servers:* Given $(f_i, g_i, [\lambda]_i)$, the servers invoke $\mathcal{F}_{\text{Zero}}$ twice to get $[z]$ and $[z']$; then, server S_j computes:
  (a) $F^j = \text{VDPF}^+.\text{Eval}(j, f_j)$ and $G^j = \text{VDPF}^+.\text{Eval}(j, g_j)$;
  (b) $[a]_j = \sum_{k=1}^{N} F_k^j$ and $[b]_j = \sum_{k=1}^{N} G_k^j$.
  (c) $\pi_j = \text{VDPF}^+.\text{Prove}(j, f_j, [z]_j)$, $\pi'_j = \text{VDPF}^+.\text{Prove}(j, g_j, [z']_j)$.
  (d) $[a_2] \leftarrow \mathcal{F}.\text{Mult}([a], [a])$.
  (e) $[\lambda'] \leftarrow \mathcal{F}.\text{Product}([\Lambda], f)$.
  (f) $[v'] \leftarrow \mathcal{F}.\text{Product}([D], f)$.

(3) S_j exchanges (h'_j, h''_j) from π'_j, π''_j, then exchanges the rest of the proofs.

(4) *Servers* halt if:
  (a) $\text{VDPF}^+.\text{Verify}(\pi_1, \pi_2, \pi_3)$ or $\text{VDPF}^+.\text{Verify}(\pi'_1, \pi'_2, \pi'_3)$ return reject.
  (b) $\mathcal{F}.\text{CheckZero}([\lambda] - [\lambda'])$ or $\mathcal{F}.\text{CheckZero}([a] - [b])$ return false.
  (c) $\mathcal{F}.\text{LTE}([a_2], [v'])$ and $\mathcal{F}.\text{LTE}([a], \sqrt{|\mathbb{F}|})$ returns false.

(5) Each server S_j updates $D^j = D^j - F^j + G^j$ and sends Ok to the client.

Figure 6: Protocol Π_{CBDC}.update.

the honest parties (the servers S_{h_1}, S_{h_2} s.t. $\{h_1, h_2, c\} = \{1, 2, 3\}$). The resulting adversarial view and the honest parties' outputs are computationally indistinguishable from those in the real execution of the protocol.

In the following we simulate the commands as if the adversary controls the client as well, and later we discuss what is changed in the simulation in case the client is honest. Whenever the simulator halts in the below description, it sends abort to the $\mathcal{F}_{\text{CBDC}}$ functionality, with which it interacts.

Register. On input $(c, \text{register})$, if $\text{ctr} = N$ send `full` to the client and halt, otherwise increment ctr and continue. Simulate $\mathcal{F}.\text{Rand}$ by computing $[\lambda] \leftarrow \text{SS.Share}_{2,3}(\lambda)$ for a uniform $\lambda \in \mathbb{F}$, hand $[\lambda]_c$ to S_c and $[\lambda]_{h_1}, [\lambda]_{h_2}$ to the client.

Read. Given, $f_{h_1}, f_{h_2}, [\lambda]_{h_1}, [\lambda]_{h_2}$, simulate $\mathcal{F}.\text{Zero}$ by computing $[z] \leftarrow \text{SS.Share}_{2,3}(0)$ and hand $[z]_c$ to S_c; then, do exactly as in the protocol, for $b \in \{1, 2\}$:

- Compute $T^{h_b} \leftarrow \text{VDPF}^+.\text{Eval}(h_b, f_{h_b})$.
- Compute $[t]_{h_b} = \sum k = 1^N T_k^{h_b}$.
- Compute $\pi_{h_b} = \text{VDPF}^+.\text{Prove}(h_b, f_{h_b}, [z]_{h_b})$, send π_{h_b} to S_c and receive π_c from S_c.

Then, simulate $\mathcal{F}.\text{Product}$ by receiving $[\Lambda]_c$ and f_c from S_c, then, halt if $[\Lambda]_c$ together with $[\Lambda]_{h_1}$ and $[\Lambda]_{h_2}$ do not form a valid Shamir sharing for a vector (note that the secrets in Λ are completely determined by $[\Lambda]_{h_1}$ and $[\Lambda]_{h_2}$ and so the secrets could not be adversarially changed), or f_c together with f_{h_1} and f_{h_2} do not form a valid point function (there exists at most one entry α at which the shared value is non-zero, and all entries form valid Shamir sharings), or reject $= \text{VDPF}^+.\text{Verify}(\pi_1, \pi_2, \pi_3)$. Otherwise (the above checks pass), compute $\tilde{\lambda} = \Lambda \cdot T$ and hand $[\tilde{\lambda}]_c$ to S_c where $[\tilde{\lambda}] \leftarrow \text{SS.Share}_{2,3}(\tilde{\lambda})$. Simulate the first instance of $\mathcal{F}.\text{CheckZero}$ by receiving $[t]_c$ from S_c, checking that it is consistent with $[t]_{h_1}, [t]_{h_2}$ computed above and verifying that $t = 1$. Similarly, simulate the second instance of $\mathcal{F}.\text{CheckZero}$ by receiving $[\lambda - \tilde{\lambda}]_c$ from S_c, checking that it is consistent with $[\lambda - \tilde{\lambda}]_{h_1}, [\lambda - \tilde{\lambda}]_{h_2}$ and verifying that $\lambda - \tilde{\lambda} = 0$. Finally, simulate $\mathcal{F}.\text{Product}$ by computing $b = D \cdot T$ and $[b] \leftarrow \text{SS.Share}_{2,3}(b)$, then sending $[b]_{h_1}, [b]_{h_2}$ to the client and $[b]_c$ to S_c.

Update. Given, $f_{h_b}, g_{h_b}, [\lambda]_{h_b}$ for $b \in \{1, 2\}$, simulate $\mathcal{F}.\text{Zero}$ by computing $[z] \leftarrow \text{SS.Share}_{2,3}(0)$, $[z'] \leftarrow \text{SS.Share}_{2,3}(0)$, and hand $[z]_c, [z']_c$ to S_c; then, do exactly as in the protocol, for $b \in \{1, 2\}$:

- Compute $F^{h_b} \leftarrow \text{VDPF}^+.\text{Eval}(h_b, f_{h_b})$ and $G^{h_b} \leftarrow \text{VDPF}^+.\text{Eval}(h_b, g_{h_b})$.
- Compute $[a]_{h_b} = \sum k = 1^N F_k^{h_b}$ and $[b]_{h_b} = \sum k = 1^N G_k^{h_b}$.
- Simulate $\mathcal{F}.\text{Mult}$ by receiving $[a]_c$ from S_c; if it is consistent with the $[a]_{h_b}$'s then compute $[a_2] \leftarrow \text{SS.Share}_{2,3}(a^2)$ and hand $[a_2]_c$ to S_c.
- Compute $\pi_{h_b} = \text{VDPF}^+.\text{Prove}(h_b, f_{h_b}, [z]_{h_b})$ and $\pi'_{h_b} = \text{VDPF}^+.\text{Prove}(h_b, g_{h_b}, [z']_{h_b})$, send (π_{h_b}, π'_{h_b}) to S_c and receive π_c, π'_c from S_c.
- Simulate $\mathcal{F}.\text{Product}$ twice by receiving $[\Lambda]_c$ and f_c from S_c, then, halt if $[\Lambda]_c$, when combined with the $[\Lambda]_{h_b}$'s, does not form a valid Shamir sharing for a vector, or f_c together with f_{h_1} and f_{h_2} do not form a valid point function, or reject is returned when computing $\text{VDPF}^+.\text{Verify}(\pi_1, \pi_2, \pi_3)$ or $\text{VDPF}^+.\text{Verify}(\pi'_1, \pi'_2, \pi'_3)$. If not halted, compute $[\lambda'] \leftarrow \text{SS.Share}_{2,3}(\Lambda \cdot F)$ and $[v'] \leftarrow \text{SS.Share}_{2,3}(D \cdot F)$, and hand $[\lambda']_c$ and $[v']_c$ to S_c.

Simulate the first instance of $\mathcal{F}.\text{CheckZero}$ by receiving $[\lambda - \lambda']_c$ from S_c, checking that it is consistent with the $[\lambda - \lambda']_{h_b}$'s and verifying that $\lambda - \lambda' = 0$. Then, simulate the second instance of $\mathcal{F}$.CheckZero by receiving $[a-b]_c$ from S_c, checking that it is consistent with the $[a-b]_{h_b}$'s computed above and verifying that $a-b=0$. Finally, simulate $\mathcal{F}$.LTE by receiving $[a_2]_c$, $[v']_c$ from S_c, checking that they are consistent with the $[a_2]_{h_b}$'s and $[v']_{h_b}$'s, and verifying that $a_2 \le v'$ (which is equivalent to verifying that $a \le D_i$). If any of the above verifications fail then halt, otherwise, update $D^{h_b} = D^{h_b} - F^{h_b} + G^{h_b}$ send Ok to the client.

The resulting view of the adversary and the output for the honest servers are perfectly simulated, that is, these views under the simulation and in the real execution of the protocol are identically distributed. We note that this is a perfect simulation even though the DPF construction is only computationally secure; this is due to the fact that when the client is corrupted then the adversary itself produces it, and so the VDPF's simulator does not come into play.

The probability that the adversary's client succeeds in submitting a malformed VDPF and still pass the verification, or pass the authentication verification without submitting a correct $\lambda = \Lambda_i$ (for some $i \in \{1, \dots, N\}$ equals in the simulation and real execution, and are both negligible in κ (the former is computationally negligible and the latter is statistically negligible).

Simulating an honest client. Here we use the VDPF's simulator (see Definition 3.1). The only difference between this case and the above (when the client is under the control of $\mathcal{A}$) is that now $\mathcal{S}$ has to produce f_c in the simulation of read (or f_c, g_c in the simulation of update). This is done by invoking the VDPF's simulator, and then sending S_c the simulated point function's share $f_c \leftarrow \mathsf{VDPF}^+.\mathcal{S}(1^\kappa, c, N)$. Combining with the rest of the simulation, the views under real execution and simulation become computationally indistinguishable (rather than identical as they were in the case the client was corrupted). It is easy to see that we can reduce the security of protocol Π to that of the VDPF^+ construction. □

5 EFFICIENT, MALICIOUS DOT-PRODUCT

The functionality $\mathcal{F}$.Product (Figure 7) is an important one in the above CBDC protocol; it receives shares of a vector $[V]$ and point functions $f = (f_1, f_2, f_3)$ (alternatively, it can receive two point functions f, g and expand g into $[V]$) from the parties, and returns the dot product of V and F, where F is the result of a full-domain evaluation of f, that is the sharings $[F] = [F_1], \dots, [F_N]$.

Setting: The functionality interacts with parties P_1, P_2, P_3 and an adversary $\mathcal{S}$.

Inputs: P_i inputs $[V]_i, f_i$, for $i \in \{1,2,3\}$.

- For $j \in \{1,2,3\}$, Expand $[F]_j \leftarrow \mathsf{VDPF.Eval}(j, f_j)$.
- For all $i \in \{1, 2, ..., N\}$:
  - $F_i \leftarrow \mathsf{SS.Reconstruct}_{1,3}([F_i])$
  - $V_i \leftarrow \mathsf{SS.Reconstruct}_{1,3}([V_i])$
  - $Z_i \leftarrow F_i \cdot V_i$.
  - Store in $[Z]$ the i-th sharing: $[Z_i] \leftarrow \mathsf{SS.Share1},3(Z_i)$.
- Wait for an input from $\mathcal{S}$, if it is $\perp$ then output $\perp$ to all parties, otherwise continue.
- Output $[Z]_j$ to P_j, for $j \in \{1,2,3\}$.

Figure 7: Functionality $\mathcal{F}$.Product

A naive implementation of $\mathcal{F}$.Product would call $\mathcal{F}$.Mult on the pairs $([V_i], [F_i])$ for every $i \in \{1, \dots, N\}$. This, however, incurs $O(N)$ communication between the parties, a cost we highly wish to avoid (otherwise the protocol could not scale well with the number of clients). Also, note that we cannot use a sum-of-products gate directly ($\mathcal{F}$.SoP), which has $O(1)$ communication in the honest majority case, because it only provides semi-honest security.

Our goal is to achieve a secure implementation of $\mathcal{F}$.Product with communication sub-linear in N, namely, with $O(\log N)$ or even constant communication. In this section we show how to do that using a new primitive we call *updatable VDPF* (or UVDPF). An updatable VDPF allows the parties who already hold some VDPF $f = (f_1, f_2, f_3)$ for some point function $F_{\alpha,\beta}$, to update the target value at entry α; that is, to produce an updated VDPF $f' = (f_1', f_2', f_3')$ for the point function $F_{\alpha,\beta'}$ for some β' that is also secret shared by them. If an implementation of that primitive can be done in sub-linear communication in N, then so can the dot product. We remark that for simplicity and better generalization, we present a protocol that only has black-box access to (2,2)-VDPFs. This protocol has $O(\log N)$ communication. For our implementation, we make an optimization that achieves $O(1)$ communication but without black-box access.

In Section 5.1 we formalize the functionality for a UVDPF, for both the $(2,2)$ and the $(2,3)$ cases. Then, in Section 5.2 we show how to use a UVDPF to implement the dot-product functionality.

5.1 Updatable $(2,3)$-VDPF

In Figure 8 we present the updatable $(2,2)$-VDPF^+ functionality, denoted $\mathcal{F}^+_{(2,2)-\mathsf{UVDPF}}$. A secure implementation of (a slightly different version of) that functionality was proposed in [55] (we note that in that paper this primitive is called 'deferred DPF').

Then, in Figure 9 we present the $(2,3)$-threshold variant of that functionality, denoted, $\mathcal{F}^+_{(2,3)-\mathsf{UVDPF}}$. Finally, using $\mathcal{F}^+_{(2,2)-\mathsf{UVDPF}}$, we construct (Fig. 10) a protocol to securely compute $\mathcal{F}^+_{(2,3)-\mathsf{UVDPF}}$.

The $(2,2)$-updatable VDPF^+ functionality (Figure 8), is a two-party functionality that, given VDPF^+ shares f_0 from the first party and f_1 from the second party, for a point function $F_{\alpha,\beta_0,\beta_1}$, and the sharings of new target values $[\beta_0'], [\beta_1']$ (these are $(2,2)$ sharings), outputs updated shares f_0' to the first party and f_1' to the second party for a new point function $F_{\alpha,\beta_0',\beta_1'}$.

Similarly, the $(2,3)$-updatable VDPF^+ functionality (Figure 9), is a *three-party* functionality that, given VDPF^+ shares f_1, f_2, f_3 from P_1, P_2, P_3, respectively, for a point function $F_{\alpha,\beta}$, and the sharings of a new target value $[\beta']$ (a Shamir $(2,3)$ sharing), outputs updated shares f_1', f_2', f_3' to point function $F_{\alpha,\beta'}$.

We note that the three-party functionality gives the adversary the opportunity to abort whereas the two-party one does not. This is due to the fact that our realization of the three-party functionality can work with a protocol for the two-party functionality that is only secure against semi-honest adversaries, as a verification is performed in the three-party protocol.

The construction begins when the servers hold shares for a $(2,3)$-VDPF, which, in our construction each share is essentially composed of two $(2,2)$-UVDPF^+ shares (note that our construction in Figure 3 uses VDPF, however, the same construction could work with an updatable VDPF). Specifically, there are two UVDPF^+s that

Setting: The functionality interacts with parties P_0 and P_1 and an adversary $\mathcal{S}$. The functionality is initialized with a VDPF$^+$ scheme.

Inputs: P_i inputs f_i, for $i \in \{0,1\}$. The parties input the $(2,2)$-sharings of $\beta'_0, \beta'_1 \in \mathbb{F}_{2^\kappa}$.

- 'Reconstruct' the function hidden by (f_0, f_1) and obtain α, β_0, β_1.
- Reconstruct the secrets β'_0 and β'_1 from their shares.
- Compute $(f'_0, f'_1) \leftarrow \mathsf{VDPF}^+(\alpha, \beta'_0, \beta'_1)$.
- Output f'_i to P_i, for $i \in \{0,1\}$.

Figure 8: Functionality $\mathcal{F}^+_{(2,2)-\mathsf{UVDPF}}$

Setting: The functionality interacts with parties P_1, P_2, P_3 and an adversary $\mathcal{S}$. The functionality is initialized with a VDPF$^+$ scheme.

Inputs: P_i inputs f_i, for $i \in \{1,2,3\}$. The parties input a $(2,3)$-Shamir sharing of $\beta' \in \mathbb{F}$.

- 'Reconstruct' $F_{\alpha,\beta}$ using (f_1, f_2, f_3) and obtain α and β.
- Reconstruct the secret β'.
- Compute $(f'_1, f'_2, f'_3) \leftarrow \mathsf{VDPF}^+(\alpha, \beta')$.
- Wait for an input from $\mathcal{S}$, if it is $\perp$ then output $\perp$ to all parties, otherwise continue.
- Output f'_i to P_i, for $i \in \{1,2,3\}$.

Figure 9: Functionality $\mathcal{F}^+_{(2,3)-\mathsf{UVDPF}}$

are already shared, namely, (f_0, f_1) encode $(\alpha, \beta_0, \beta_1)$ and (g_0, g_1) encode $(\alpha, \gamma_0, \gamma_1)$. Recall that each party P_i ($i \in \{1,2,3\}$) has shares of two $(2,2)$-UVDPF$^+$s, that is, P_1 has (f_0, g_0), P_2 has (f_1, g_0) and P_3 has (f_1, g_1). It holds that $(\beta_0 \oplus \gamma_0), (\beta_1 \oplus \gamma_0) \odot 2, (\beta_1 \oplus \gamma_1) \odot 3$ form a valid Shamir sharing of β.

Let δ be the value that the parties wish to plug at location α instead of β, we assume that the parties hold the $(2,3)$-sharing $[\delta]$. The parties' goal is to obtain the sharings $[\beta'_0], [\beta'_1], [\gamma'_0], [\gamma'_1]$ s.t.

$$[\delta]_1 = (\beta'_0 \oplus \gamma'_0),\ [\delta]_2 = (\beta'_1 \oplus \gamma'_0) \odot 2, [\delta]_3 = (\beta'_1 \oplus \gamma'_1) \odot 3 \quad (3)$$

form a valid Shamir sharing of δ, and hand those sharings to the two instances of $\mathcal{F}_{(2,2)-\mathsf{UVDPF}^+}$. That is, the first (f_0, f_1) should be updated with $[\beta'_0], [\beta'_1]$ and (g_0, g_1) should be updated with $[\gamma'_0], [\gamma'_1]$. In protocol $\Pi_{(2,3)-\mathsf{UVDPF}^+}$ (Figure 10) the parties obtain the appropriate $(2,2)$-sharings of $[\beta'_0], [\beta'_1], [\gamma'_0], [\gamma'_1]$ that are later fed to the two instances of $\mathcal{F}_{(2,2)-\mathsf{UVDPF}^+}$ and obtain the new shares of the $(2,3)$-VDPF$^+$.

5.2 The Dot-Product Protocol

Given a $(2,3)$-UVDPF, the construction of our maliciously secure dot-product is presented in Figure 11. It accepts a vector and a DPF, expands the DPF, and computes a sharing of their dot product. We note that the protocol relies on an additional functionality, $\mathcal{F}$.SoP for computation of sum-of-products. We note that $\mathcal{F}$.SoP only needs to be a semi-honest functionality that allows the adversary to inject an additive error to the result, but this is not of concern in our larger protocol as we achieve the MAC'ed result using another invocation and compare the results. Since the MAC value is random and independent, the adversary cannot inject additive errors to both results such that they match (i.e., one is a MAC'ed version of the

Inputs: P_i inputs f_i, for $i \in \{1,2,3\}$. The parties input a $(2,3)$-Shamir sharing of $\beta' \in \mathbb{F}$.

Protocol: If, at any of the following steps, any party receives $\perp$ from a functionality invocation, then it aborts.

(1) Pick random bit sharings:
   - $([\beta'_{0,\kappa-1}], \dots, [\beta'_{0,0}])$.
   - $([\gamma'_{0,\kappa-1}], \dots, [\gamma'_{0,0}])$.

(2) For every $i \in [0, \kappa-1]$, compute $[\delta_{1,i}] = [\beta'_{0,i}] \oplus [\gamma'_{0,i}]$.

(3) Compute: $[\delta_1] = \sum_{i=0}^{\kappa-1} 2^i \cdot [\delta_{1,i}]$.
Note that $\delta_1 = (\beta'_0 \oplus \gamma'_0)$ is S_1's share; together with δ, they completely define the degree-1 polynomial $P(\mathsf{x}) = \delta + a\mathsf{x}$ s.t. $P(1) = \delta_1 = \delta + a$, meaning that $a = \delta - \delta_1$. Given δ and a, it follows that $\delta_2 = P(2) = \delta + 2a$ and $\delta_3 = P(3) = \delta + 3a$. Thus:

(4) Compute $([\delta_{2,\kappa-1}], \dots, [\delta_{2,0}]) \leftarrow \mathcal{F}.\mathsf{A2B}([\delta_2])$ and $([\delta_{3,\kappa-1}], \dots, [\delta_{3,0}]) \leftarrow \mathcal{F}.\mathsf{A2B}([\delta_3])$, where $[\delta_2] = [\delta] + 2[a]$ and $[\delta_3] = [\delta] + 3[a]$.

(5) For every $i \in [0, \kappa-1]$, compute $[\beta'_{1,i}] = [\delta_{2,i}] \oplus [\gamma'_{0,i}]$ and $[\gamma'_{1,i}] = [\beta'_{1,i}] \oplus [\delta_{3,i}]$.

(6) Generate random sharings of bits, denoted $[x_i], [y_i], [z_i], [w_i]$ for every $i \in [0, \kappa-1]$.

(7) Open to S_1 the values x_i, y_i, z_i, w_i, for every $i \in [0, \kappa-1]$. Denote $x = (x_{\kappa-1}, \dots, x_0)$ and similarly for y, z, w.

(8) Open to S_3 the value $x_i \oplus \beta'_{0,i}$, $y_i \oplus \beta'_{1,i}$, $z_i \oplus \gamma'_{0,i}$, $w_i \oplus \gamma'_{1,i}$. Denote $x' = (x_{\kappa-1} \oplus \beta'_{0,\kappa-1}, \dots, x_0 \oplus \beta'_{0,0})$ and similarly y', z', w'.

(9) S_1 and S_3 call the updatable VDPF$^+$ functionality twice:
   (a) S_1 inputs f_0, x, y and S_3 inputs f_1, x', y'. S_1 receives f'_0 and S_3 receives f'_1.
   (b) S_1 inputs g_0, z, w and S_3 inputs g_1, z', w'. S_1 receives g'_0 and S_3 receives g'_1.

(10) S_1 sends g'_0 to S_2 and S_3 sends f'_1 to S_2.

(11) The three servers obtain $[r] \leftarrow \mathcal{F}.\mathsf{Rand}$; then they run Prove and Verify on the new shares $(f'_0, g'_0), (f'_1, g'_0)$ and (f'_1, g'_1) using their shares of $[r]$.

Figure 10: Protocol $\Pi_{(2,3)-\mathsf{UVDPF}^+}$

other). This technique was used in previous works for maliciously secure MPC (e.g., [16]).

6 IMPLEMENTATION, EVALUATION AND APPLICATIONS

In this section, we implement prototypes of our VDPF constructions and a basic three-server ORAM scheme that builds on top of it. We then use these tools to construct our private Π_{CBDC} protocol presented in Figures 4, 5, 6. Our code is written in C++ and consists of approximately 7,000 lines of new code[5]. Our DPF implementation leverages the highly efficient (2,2)-DPF implementation from [32], but we extend their code[6] to allow for larger than 1-bit DPFs.

We ran all benchmarks on a single Azure Standard E32s (v5) VM server, which has 16 physical cores (32 vCPUs) and 256GB of RAM. We simulated network latency and bandwidth using the *tc* command.

[5]https://github.com/guyz/abcledger
[6]https://github.com/dkales/dpf-cpp

Inputs: P_i has V^i and f_i as inputs, for $i \in \{1, 2, 3\}$, where V^i are shares of a vector and f_i are shares for a VDPF. We assume that the VDPF was already proven valid.

Protocol:

(1) The parties invoke $[m] \leftarrow \mathcal{F}.\text{Rand}()$.
(2) Party P_i computes $A^i = \text{VDPF.Eval}(i, f_i)$ and then $[a]_i = \sum_{k=1}^{N} A^i_k$.
(3) The parties compute $[ma] \leftarrow \mathcal{F}.\text{Mult}([m], [a])$.
(4) The parties compute $g' \leftarrow \mathcal{F}_{(2,3)-\text{UVDPF}}(f, [ma])$, which results with shares f'_1, f'_2, f'_3 for a point function $F_{\alpha,ma}$.
(5) Party P_i computes $B^i = \text{VDPF.Eval}(i, f'_i)$ and then $[b]_i = \sum_{k=1}^{N} B^i_k$.
(6) The parties compute $[va] \leftarrow \mathcal{F}_{\text{SoP}}([V], [A])$ and $[vb] \leftarrow \mathcal{F}_{\text{SoP}}([V], [B])$.
(7) Compute $[vam] \leftarrow \mathcal{F}.\text{Mult}([va], [m])$.
(8) Call $\mathcal{F}.\text{CheckZero}([vam] - [vb])$, if the result is false then halt, otherwise output $[va]$.

Figure 11: Protocol Π_{Product}

In addition, to further illustrate the applicability of our VDPF construction, we sketch in Section 6.3 how it can be extended to the application of building a Distributed ORAM (DORAM), and qualitatively compare it with the state-of-the-art.

6.1 DPFs

We implement both verifiable and non-verifiable versions of the state-of-the-art (2,2)-DPF from [7], and our two DPF constructions (Sections 3.2, A.1). We also compare analytically to the (2,3)-DPF from [11].

For all of these constructions, we measure the time it takes to evaluate the entire domain for different domain sizes (shown in Figure 12), as well as key sizes (in Table 2). As expected, our constructions have key sizes that are 2x bigger than the baseline[7], but this is marginal even for very large domains ($\sim 2KB$ keys for $N = 2^{50}$). Similarly, for evaluation, our (2,3)-DPF construction is only 2x slower than the baseline, and similarly, the VDPF construction has an additional estimated 2x overhead, since we are effectively evaluating another level.

Our (2,3)-DPF from Section A.1 is $\sim$ 2.2× faster for $n > 2^{20}$ compared to a (2, 2)-DPF. We estimate that this gap will increase (in our favor) using GPUs, for two reasons: (i) AES is hardware-accelerated in CPU, but not in GPU; (ii) In contrast, GPUs are massively parallel and can perform vector operations well.

	2^{20}	2^{30}	2^{40}	2^{50}
(2,2)-VDPF [7]	357	537	717	897
(2,3)-VDPF [11] (Analytical)	51024	120804	188664	289824
(2,3)-VDPF (Section 3.2)	850	1210	1570	1930
VDPF (Sublinear) (Section A.1)	980	1340	1700	2060

Table 2: Comparing key sizes (in bytes) of our constructions compared to the baseline (2,2) DPF of [6] and (2,3) DPF of [11]. Since [11] does not provide an implementation, we provide an analytical estimate.

[7]A (2,3) DPF has two (2,2) keys, and the DPF with sublinear PRF calls has 4 half-sized such keys.

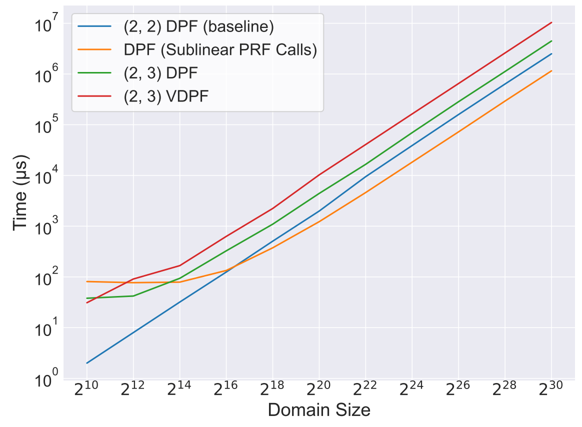


Figure 12: Full domain evaluation of the various DPF constructions, compared to the baseline (2,2) DPF.

6.2 Account-based Privacy-preserving Cryptocurrency and CBDC

We now benchmark our main CBDC protocol. Our results support our hypothesis that we can scale to large anonymity sets, including those exceeding a million accounts. This vastly surpasses previous systems, which only provided a k-anonymity set between 10-256. We benchmark our system against Solidus [14], the closest existing model, which also uses an ORAM for privacy in an account-based ledger and marks the current state-of-the-art.

We conducted end-to-end transaction tests comparing Solidus and our protocols (semi-honest and malicious $\Pi_{CBDC}.Update$ protocols). To create a fair comparison, we optimized Solidus parameters to leverage all available cores and system memory. Our findings, detailed in Figure 13, reveal that our protocols significantly outperform Solidus in transaction throughput and memory efficiency for up to $N = 2^{18}$ accounts, and continue to outperform it up to $N = 2^{22}$ accounts. While Solidus manages to close the gap at $N = 2^{24}$ accounts, it fails to offer complete anonymity like our system and it is less efficient in memory usage. Beyond $N = 2^{24}$, Solidus exhausts memory, whereas our protocols remain scalable.

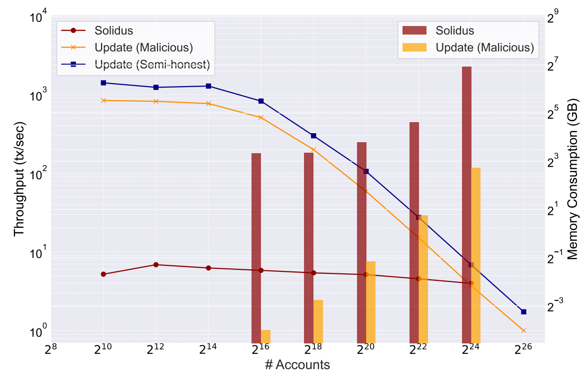


Figure 13: Transaction throughput and memory usage. Side-by-side comparison with Solidus.

Additionally, we believe our system has the potential for horizontal scaling across multiple servers, as it exhibits minimal bandwidth requirements and DPF evaluations and inner product computations (which are the bottlenecks) could be run in parallel. Furthermore, end-to-end latency, as shown in Figure 14, remains low across various network conditions, with acceptable delays even on slower networks.

6.3 Three-party Distributed ORAM (DORAM)

A closely related problem to an ORAM scheme, which our CBDC protocol above implements, is that of a Distributed ORAM (DORAM). DORAM is a fundamental building block in MPC, as it allows securely running RAM programs directly, as opposed to converting them first to circuits, which can yield meaningful performance gains. Thus, a significant body of research was dedicated to optimizing DORAM for the two-party and three-party cases [8, 10, 11, 23, 25, 43, 48, 56], with three-party DORAMs being the most efficient. There are two major branches of DORAMs in the literature: (i) those based on classical sublinear-computation ORAM constructions, with logarithmic overhead (e.g., [25, 43]), or with square-root overhead (e.g., [8]); (ii) and those based on DPFs [10, 11, 23, 48, 56]. While the latter require linear computation, they are extremely lightweight, having sublinear communication and a very low number of rounds in comparison. For that reason, they tend to scale better for mid-to-large memory sizes (e.g., up to 2^{26} records [23, 56]), or are suited for higher-latency environments, for example when the parties communicate over the internet and are not co-located in the same data center.

Given our (2,3)-VDPF construction, our work fits in the second bucket of research targeting DPF-based DORAMs. We have already shown how our construction enables building an efficient three-server ORAM. To turn it into a DORAM, we also need the servers to generate the VDPF in MPC, since there is no client. We illustrate this process in the following protocol sketch which builds a DORAM from our DPF construction.

6.3.1 *(2,3)-VDPF DORAM protocol:*

- **Input.** The three parties start with a shamir-sharing over $\mathbb{F}_{2^l}$ of the secret DPF parameters $[\alpha]$, $[\beta]$. As shown in Figure 3, Line 1 of VDPF.Gen, the parties need to obtain shares (of shares) of β which they can achieve through a constant-round re-sharing protocol.
- Recall that in Figure 3, VDPF.Gen constructs two underlying VDPF+ instances, each of which is composed of a single two-party VDPF with auxiliary data z. Generating a VDPF in a distributed manner can be done with the well-known protocol of [23], and it takes $O(logN)$ rounds and server communication and $O(N)$ computation. After this, the parties have generated two VDPFs and have evaluated them at the same time over the entire domain.
- However, in order to turn these into VDPFs+, we need to handle the auxiliary data $z \leftarrow \beta_0 \oplus \mathsf{VDPF.Eval}(0, f_0, \alpha)$. The parties already have a sharing of β_0, but to obtain $\mathsf{VDPF.Eval}(0, f_0, \alpha)$, they need to somehow privately evaluate each VDPF at the point α. recall that P1, P2 both have the first key of the second DPF, and similarly P2, P3 have the second key of the first DPF. This redundancy allows the third party to perform a PIR query to privately obtain the necessary value. For example, for the second VDPF, P3 can read $\mathsf{VDPF.Eval}(0, f_0, \alpha)$ by generating a DPF (locally) and performing a 2-server PIR to read the value at α from P1 and P2. Because P3 does not actually know α, P1 and P2 first shift α by some randomness r and open the result towards P3. They also shift their own vector by r positions locally, ensuring that P3 reads the correct value. Treatment of the other DPF is symmetrical. Note that this takes only a (small) constant number of rounds, $O(logN)$ communication and $O(N)$ computation, so it does not significantly add to the protocol's overhead.
- To complete the evaluation of the (2,3)-VDPF, the parties perform local operations and obtain their final share. With this, they can perform either a private read or write over the secret-shared memory, in the same way we described for our ORAM scheme.

Overall, this protocol has $O(logN)$ communication and round-complexity, and most of the overhead is in the distributed generation of [23], which is already quite efficient and has small constants.

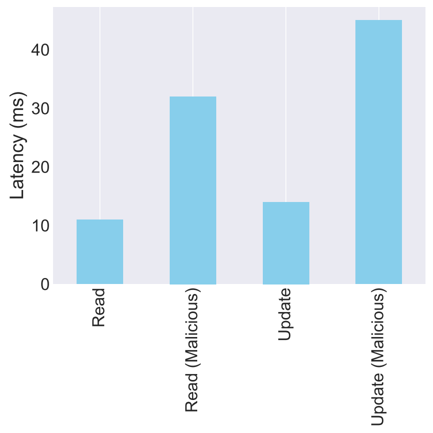
Comparing to the state-of-the-art. Several recent works tried to construct an efficient three-party DORAM using DPFs. In [10, 11], the authors construct three-party DPFs that are significantly less efficient than ours. Therefore, their overall communication overhead between the servers is worse. In the first, communication is $O(\sqrt{N})$ per-query (but with a constant number of rounds), while in the latter, it is $O(log^2 N)$ for both communication and rounds, with high constants. More similar to our work, [48, 56] presented a $O(logN)$ communication/rounds protocol for a three-party DORAM, but their protocol is in the server-aided model and therefore not a true three-party protocol. We thus estimate that a DORAM protocol based on our VDPF would yield the most efficient three-party DPF-based DORAM protocol to date, and that compared to other DORAMs, ours would be the most efficient for mid-sized memories or in high-latency environments. We leave implementing and benchmarking such a system for future work.

ACKNOWLEDGMENTS

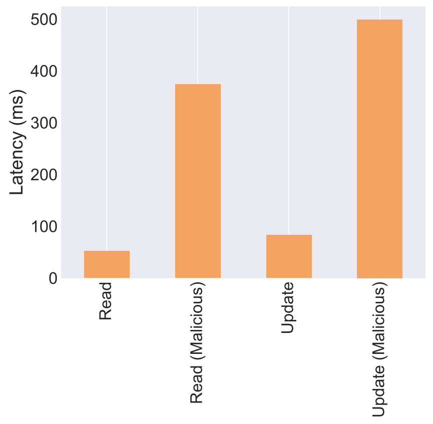
We wish to thank Itzik Grossman for help with optimizing the code.

REFERENCES

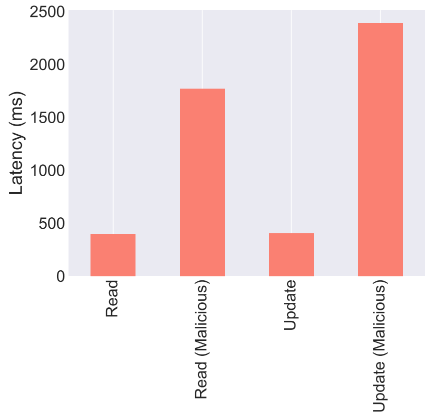
[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. 2020. Blinder–Scalable, Robust Anonymous Committed Broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* 1233–1252.
[2] Toni Ahnert, Peter Hoffmann, and Cyril Monnet. 2022. The digital economy, privacy, and CBDC. (2022).
[3] Elli Androulaki, Marcus Brandenburger, Angelo De Caro, Kaoutar Elkhiyaoui, Alexandros Filios, Liran Funaro, Yacov Manevich, Senthilnathan Natarajan, and Manish Sethi. 2023. A Framework for Resilient, Transparent, High-throughput, Privacy-Enabled Central Bank Digital Currencies. *Cryptology ePrint Archive* (2023).
[4] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2021. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 762–776.
[5] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 947–964.
[6] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques.* Springer, 337–367.
[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 1292–1303.



(a) RTT = 1ms



(b) RTT = 10ms



(c) RTT = 100ms

Figure 14: Latency in processing a request under different channel latency for $N = 2^{20}$. Requests can be parallelized so this is indicative, and bandwidth changes had no effect thanks to the constant communication complexity.

[8] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. 2023. Ramen: Souper fast three-party computation for ram programs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3284–3297.

[9] Markus K BRUNNERMEIER and Jean-Pierre Landau. 2022. The digital euro: policy implications and perspectives. (2022).

[10] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. 2020. Efficient 3-party distributed ORAM. In *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings 12*. Springer, 215–232.

[11] Paul Bunn, Eyal Kushilevitz, and Rafail Ostrovsky. 2022. CNF-FSS and its applications. In *IACR International Conference on Public-Key Cryptography*. Springer, 283–314.

[12] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*. Springer, 423–443.

[13] Matteo Campanelli and Mathias Hall-Andersen. 2022. Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 652–666.

[14] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. 2017. Solidus: Confidential distributed ledger transactions via PVORM. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 701–717.

[15] Panagiotis Chatzigiannis and Foteini Baldimtsi. 2021. Miniledger: compact-sized anonymous and auditable distributed payments. In *European Symposium on Research in Computer Security*. Springer, 407–429.

[16] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO*, Hovav Shacham and Alexandra Boldyreva (Eds.).

[17] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*. 259–282.

[18] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 321–338.

[19] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. 2013. Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*. Springer, 1–18.

[20] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. 2022. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2450–2468.

[21] Leo de Castro and Anitgoni Polychroniadou. 2022. Lightweight, maliciously secure verifiable function secret sharing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 150–179.

[22] Benjamin E Diamond. 2021. Many-out-of-many proofs and applications to anonymous zether. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1800–1817.

[23] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 523–535.

[24] Shlomi Dolev and Ziyu Wang. 2020. SodsMPC: FSM based anonymous and private quantum-safe smart contracts. In *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. IEEE, 1–10.

[25] Brett Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. 2023. GigaDORAM: breaking the billion address barrier. In *Proceedings of the 32nd USENIX Conference on Security Symposium*. 3871–3888.

[26] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. 2019. Quisquis: A new design for anonymous cryptocurrencies. In *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part I 25*. Springer, 649–678.

[27] Georg Fuchsbauer, Michele Orrù, and Yannick Seurin. 2019. Aggregate cash systems: A cryptographic investigation of mimblewimble. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*. Springer, 657–689.

[28] Niv Gilboa and Yuval Ishai. 2014. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*. Springer, 640–658.

[29] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[30] Nerla Jean-Louis, Yunqi Li, Yan Ji, Harjasleen Malvai, Thomas Yurek, Sylvain Bellemare, and Andrew Miller. 2023. Sgxonerated: Finding (and partially fixing) privacy flaws in tee-based smart contract platforms without breaking the tee. *Cryptology ePrint Archive* (2023).

[31] Georgios Kaissis, Alexander Ziller, Jonathan Passerat-Palmbach, Théo Ryffel, Dmitrii Usynin, Andrew Trask, Ionésio Lima Jr, Jason Mancuso, Friederike Jungmann, Marc-Matthias Steinborn, et al. 2021. End-to-end privacy preserving deep learning on multi-institutional medical imaging. *Nature Machine Intelligence* 3, 6 (2021), 473–484.

[32] Daniel Kales, Olamide Omolola, and Sebastian Ramacher. 2019. Revisiting user privacy for certificate transparency. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 432–447.

[33] Yunqi Li, Kyle Soska, Zhen Huang, Sylvain Bellemare, Mikerah Quintyne-Collins, Lun Wang, Xiaoyuan Liu, Dawn Song, and Andrew Miller. 2023. Ratel: MPC-extensions for Smart Contracts. *Cryptology ePrint Archive* (2023).

[34] James Lovejoy, Anders Brownworth, Madars Virza, and Neha Narula. [n. d.]. PARSEC: Executing Smart Contracts in Parallel. ([n. d.]).

[35] James Lovejoy, Madars Virza, Cory Fields, Kevin Karwaski, Anders Brownworth, and Neha Narula. 2023. Hamilton: A {High-Performance} Transaction Processor for Central Bank Digital Currencies. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 901–915.

[36] Steve Lu and Rafail Ostrovsky. 2013. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference*. Springer, 377–396.

[37] Varun Madathil and Alessandra Scafuro. 2023. PriFHEte: Achieving Full-Privacy in Account-based Cryptocurrencies is Possible. *Cryptology ePrint Archive* (2023).

[38] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. 2013. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 397–411.
[39] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. 27–30.
[40] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, et al. 2017. An empirical analysis of traceability in the monero blockchain. *arXiv preprint arXiv:1704.04299* (2017).
[41] Neha Narula, Willy Vasquez, and Madars Virza. 2018. {zkLedger}:{Privacy-Preserving} auditing for distributed ledgers. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 65–80.
[42] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. 2021. Spectrum: High-Bandwidth Anonymous Broadcast with Malicious Security. *IACR Cryptol. ePrint Arch.* 2021 (2021), 325.
[43] Daniel Noble, Brett Hemenway Falk, and Rafail Ostrovsky. 2024. MetaDORAM: Breaking the Log-Overhead Information Theoretic Barrier. *Cryptology ePrint Archive* (2024).
[44] Peterson K Ozili. 2022. Central bank digital currency in Nigeria: opportunities and risks. In *The new digital era: digitalisation, emerging risks and opportunities*. Emerald Publishing Limited, 125–133.
[45] Alejandro Ranchal-Pedrosa and Vincent Gramoli. 2019. Platypus: a partially synchronous offchain protocol for blockchains. *arXiv preprint arXiv:1907.03730* (2019).
[46] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2020. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *arXiv preprint arXiv:2006.04593* (2020).
[47] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*. IEEE, 459–474.
[48] Sajin Sasy, Adithya Vadapalli, and Ian Goldberg. 2023. PRAC: Round-Efficient 3-Party MPC for Dynamic Data Structures. *Cryptology ePrint Archive* (2023).
[49] Sacha Servan-Schreiber, Simon Beyzerov, Eli Yablon, and Hyojae Park. 2023. Private access control for function secret sharing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 809–828.
[50] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. 2022. Private approximate nearest neighbor search with sublinear communication. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 911–929.
[51] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.
[52] Chan Wang Mong Tikvah. 2023. A Privacy-preserving Central Bank Ledger for Central Bank Digital Currency. *Cryptology ePrint Archive* (2023).
[53] Alin Tomescu, Adithya Bhat, Benny Applebaum, Ittai Abraham, Guy Gueta, Benny Pinkas, and Avishay Yanai. 2022. Utt: Decentralized ecash with accountable privacy. *Cryptology ePrint Archive* (2022).
[54] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. 2020. Epione: Lightweight contact tracing with strong privacy. *arXiv preprint arXiv:2004.13293* (2020).
[55] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. 2021. You May Also Like... Privacy: Recommendation Systems Meet PIR. *Proc. Priv. Enhancing Technol.* 2021, 4 (2021), 30–53.
[56] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. 2023. Duoram: A Bandwidth-Efficient Distributed ORAM for 2-and 3-Party Computation. In *32nd USENIX Security Symposium*.
[57] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861.
[58] Kilian Wenker. 2022. Retail central bank digital currencies (CBDC), Disintermediation and financial privacy: The case of the Bahamian sand dollar. *FinTech* 1, 4 (2022), 345–361.
[59] Jianguo Xu. 2022. Developments and implications of central bank digital currency: The case of China e-CNY. *Asian Economic Policy Review* 17, 2 (2022), 235–250.
[60] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting square-root ORAM: efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 218–234.
[61] Guy Zyskind, Oz Nathan, et al. 2015. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE security and privacy workshops*. IEEE, 180–184.
[62] Guy Zyskind, Oz Nathan, and Alex Pentland. 2015. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).
[63] Guy Zyskind, Tobin South, and Alex Pentland. 2023. Don't forget private retrieval: distributed private similarity search for large language models. *arXiv preprint arXiv:2311.12955* (2023).

A APPENDIX

A.1 A VDPF with sublinear PRF calls

Another application of our (2,3)-VDPF construction is to build, in a black-box manner, a VDPF with sublinear PRF calls (the main computational bottleneck in evaluating DPFs). Note that in this construction, calling Eval on x produces a degree-2 sharing of $y = F(x)$. This has two implications: first, is that this construction can still withstand malicious clients (due to verifiability), but only semi-honest servers; second, we lose our one *free* multiplication, so we can no longer combine PIR with PIW in a simple way, similar to (2,2) VDPFs. In other words, this DPF is a potential (faster) drop-in replacement for applications using (2,2) VDPFs (e.g., [17, 18, 42, 50]), designed for the three-party model.

The full construction is detailed in the full version of this paper. We only illustrate the main idea here, which is to re-interpret each DPF as a square matrix (instead of a vector), where α is defined by $\alpha_{row}, \alpha_{col}$. Then, for a DPF with domain of size $n = |D|$, we can share two DPFs that have a domain of size $\sqrt{n}$ instead. One for the row and one for the column. A similar idea was used in prior works (e.g. [1, 11]). To perform a full-domain evaluation efficiently, we can first run a full-domain evaluation on the row and column DPFs. We then obtain two one-hot-vectors, which we use to expand the full DPF. This is done by taking each value in the row vector, and multiplying it with each value in the column vector. It is easy to see that the result is a sharing of a vector $\boldsymbol{y^n}$, with a sharing of β in index $\alpha_{row} \cdot \sqrt{n} + \alpha_{col} = \alpha$, and 0 everywhere else.

A.2 Running in Bank Mode

There are several practical ways to deploy our CBDC protocol. As mentioned, one method is to have the Central Bank distribute the maintenance of the system to three trust zones (e.g., with two other independent organizations).

Another method is to further distribute the system across commercial banks. Similar to the bank-to-bank model employed by [14, 15, 41], we could ask each bank to instantiate its own version of the protocol for its own users. For each instantiation, the commercial bank would be one of the three parties, and the two other parties will be played by other independent organizations (e.g., the Central Bank and an indepdendent non-profit).

In this deployment scenario, a user reading its balance will contract its own bank's database obliviously. Unlike similar works, in our scenario the bank will remain oblivious to which user interacts with it. To send a transaction, a user will split its update into two parts (decreasing her own amount and increasing the receiving party's amount). One update will be performed on the sending user's bank ORAM, and similarly for the receiving user's bank ORAM. The two banks will 'compare notes' and make sure that the verification procedures all pass, but will otherwise remain oblivious.